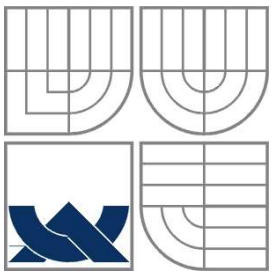
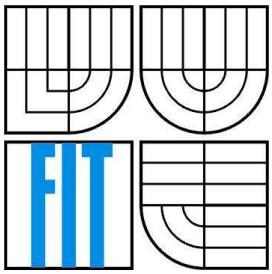


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MODEL ÚLOH S OMEZENÍMI A MECHANISMŮ JEJICH PLÁNOVANÍ V UPPAAL SMC

MODEL OF TASKS WITH CONSTRAINS AND MECHANISMS OF THEIR SCHEDULING IN UPPAAL SMC

BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

FILIP PALÚCH

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JOSEF STRNADEL, Ph.D.

BRNO 2016

Abstrakt

Cieľom je vytvorenie prehľadu mechanizmov plánovania úloh v jednoprocesorovom prostredí. Požitím týchto mechanizmov sú navrhnuté a implementované jednotlivé modely v nástroji UPPAAL, na ktorý sa táto práca orientuje. Výsledkom práce je overenie validity a vlastností mechanizmov získaných na základe implementovania modelov. Na porovnávanie výsledkov z nástroja UPPAAL sú využíté nástroje TimesTool a Cheddar.

Abstract

The effort of this thesis is the review of mechanisms of tasks planning in singlecore environment. Each models are designed and implemented in tool UPPAAL using these mechanisms. The main focus in this thesis is the tool UPPAAL. Result of this thesis is verification of each mechanisms properties received from implementation of models in UPPAAL TimesTool and Cheddar are the tools which are used for comparing our results from UPPAAL.

Klíčová slova

Model, Úloha, Plánovanie, UPPAAL, Časovaný automat, Analýza, Verifikácia, Štatistické overovanie modelov

Keywords

Model, Task, Planning, UPPAAL, Timed Automaton, Analysis, Verification, Statistical model checking

Citace

Palúch Filip: Model úloh s omezeními a mechanismů jejich plánovaní v UPPAAL SMC, bakalářská práce, Brno, FIT VUT v Brně, 2016

Model úloh s omezeními a mechanismů jejich plánovaní v UPPAAL SMC

Prohlášení

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Josefa Strnadela, Ph.D. Uviedol som všetky literárne pamene a publikácie, z ktorých som čerpal.

……
Filip Palúch
17.5.2016

Poděkování

Chcel by som poďakovať všetkým, ktorí mi pomohli pri vytváraní bakalárskej práce a získavaní zdrojov potrebných na jej písanie. Predovšetkým by som chcel poďakovať môjmu vedúcemu Ing. Josefovi Strnadelovi, Ph.D., ktorý mi poskytol veľmi hodnotné rady a skúsenosti v tomto smere. Bez jeho pomoci by vytvorenie tejto práce nebolo možné.

Obsah

1 Úvod

Modelovanie a simúlacie sú témou, ktoré sú súčasťou nášho každodenného života. Modely systémov, ktoré sú zväčša vytvorené na simulovanie za účelom získania nejakých výsledkov alebo dokázania nejakých faktov sú v skutočnosti prezentované reálnymi situáciami vo svete. Model nejakého reálneho systému si vieme predstaviť napríklad model križovatky, model príchodu vlaku na železničné priecestie alebo model jadrovej elektrárne. Na niektorých modeloch, vieme simulovať situácie, ktoré môžu mať až fatálne následky, v prípade výskytu nejakej chyby alebo úplného zlyhania systému. Niekedy sú simulácie modelov jediným možným spôsobom ako vyriešiť nejaký problém, pretože v reálnom svete nie sme schopný túto situáciu dosiahnuť v nejakom reálnom čase alebo ju nevieme dosiahnuť vôbec.

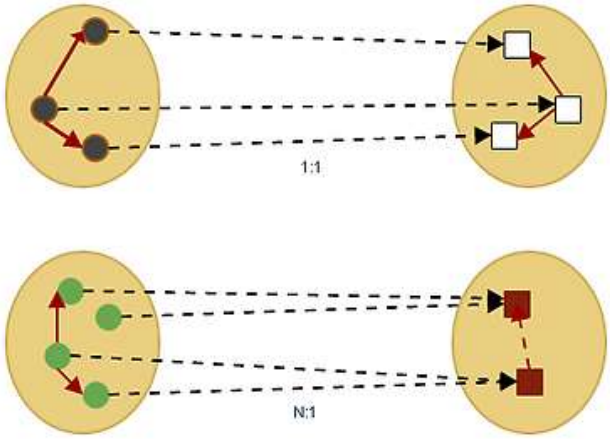
V tomto texte sa najskôr zameriame na vysvetlenie základných pojmov a výrazov, ktoré sa týkajú simulácií a modelovania, aby bol čitateľovi text jasný a zrozumiteľný. Následne sa zameriame na nástroj UPPAAL a jeho rozšírenie UPPAAL SMC, na ktorý je táto práca zameraná. Popíšeme si načo tento nástroj slúži a taktiež na čo všetko sa dá využiť. Na to, aby sme ho mohli využívať v praxi na vytváranie modelov, si vysvetlíme základné konštrukcie, ktoré sú v rámci nástroja UPPAAL využívané. Taktiež si ukážeme ako modely implementované v tomto nástroji overovať. To nám môže pomáhať pri odhaľovaní chýb v systéme. Samostatná kapitola bude venovaná predstaveniu mechanizmov, podľa ktorých môžu byť jednotlivé úlohy plánované. Po prečítaní tejto kapitoly by mal čitateľ pochopiť základné princípy ako tieto mechanizmy fungujú. V ďalšej časti textu sa budeme venovať návrhu a implementácii modelov, plánovaných jednotlivými mechanizmami. Poslednú časť textu venujeme testovaniu a porovnaniu výsledkov, ktoré sme dostali z nástroja UPPAAL, s výsledkami iných nástrojov, ktoré budú slúžiť na porovnanie. Na základe týchto výsledkov sa budeme snažiť dokázať validitu a vlastnosti jednotlivých modelov, plánovaných na základe mechanizmov, ktoré boli vysvetlené v texte.

2 Zhrnutie základných pojmov

2.1 Model

Samotný model by sme mohli definovať ako napodobneninu systému iným systémom pričom sa snažíme v rámci modelu zachovať vlastnosti, ktoré chceme skúmať.(2) Pri tvorbe modelu sa väčšinou nezameriavame na všetky vlastnosti systému, ale iba na vlastnosti, ktoré sú pre funkčnosť daného modelu najdôležitejšie. Podľa úrovne abstrakcie môžeme modely rozdeliť na (2):

- Izomorfné – tento model nezanedbáva žiadne vlastnosti modelovaného systému (1:1)
- Homomorfné – abstrahuje vlastnosti modelovaného systému, ktoré pre daný model nie sú dôležité (N:1)



Obrázok 2.1: Príklad izomorfného a homomorfného modelu

Príkladom takéhoto homomorfného modelu by mohol byť model križovatky s autami.

2.2 Modelovanie

Modelovanie patrí medzi neodlučiteľné súčasti života človeka od čias histórie až po súčasný svet. Je súčasťou každodenného života a vytvára v mysli človeka obraz reálneho sveta. Pri vytváraní modelu využívame vlastné znalosti, ktoré sú niečím podložené. Model je výsledkom abstrahovania reality skutočného sveta pričom výsledkom by mal byť pohľad na systém, z ktorého sme ho chceli skúmať. Modelovať sme schopný iba systémy, o ktorých máme dostatočné množstvo informácií. (2)

2.3 Simulácia

Simulácia je získavanie nových znalostí o systéme experimentovaním s jeho modelom. (1) Simulácia nám vytvorí výsledky, ktoré boli získané z vytvoreného modelu. Základnou charakteristikou modelu je schopnosť vykonávania danej simulácie na počítači. Z výsledku experimentov sme schopný určiť korektnosť vytvoreného modelu a jeho podobnosť so správaním

systému v reálnom svete. V niektorých prípadoch je experimentovanie s modelom jediným možným riešením nakoľko simulácia systému v reálnom svete nemusí byť možná či už z bezpečnostného, finančného hľadiska alebo vzhľadom na celkovú dĺžku trvania celej simulácie.(5)

Počítačové systémy plnia v oblasti modernej vedy veľmi významnú úlohu, a ich prípadné chyby môžu mať dramatické následky. Z tohto dôvodu je vynakladaná značná aktivita a úsilie na opravu týchto chýb. Riešenie týchto problémov je hlavným predmetom záujmu či už na akademickej úrovni alebo v oblasti priemyslu. Medzi známe spôsoby detekovania chýb patrí testovanie systémov na základe istých testovacích prípadov. Druhým spôsobom odhaľovania chýb sú formálne metódy, kde na základe sledovania modelu, môžeme garantovať absenciu chýb v danom systéme. (3)

2.4 Overovanie systémov

V tejto podkapitole sme použili informácie zo zdrojov (3), (4). Formálna verifikácia bola spočiatku spätá so správaním softwarových a hardwarových systémov na diskrétnej úrovni. Až posledné roky ukázali, že hlavnú úlohu v procese verifikácie budú hrať aspekty týchto systémov v reálnom čase. Vývoj formálnych techník pre systémy sa stal hlavným subjektom intenzívnych štúdií. (3)

Výsledkom vývoja teórie automatov sa stal nástroj UPPAAL, ktorý sa stal lídrom v danej oblasti. Cieľom celého vývoja bolo prevedenie techník sledovania reálnych systémov na modely. Tieto modely sú schopné zachytiť správanie reálneho systému závislého na hodinách, ktoré môžu byť resetované.

Konečná kvalita modelu sa odvíja od našich znalostí pri samotnom vytváraní modelu a to má vplyv aj na výsledky, ktoré vzniknú testovaním vyrobeného modelu. Samotné výsledky testovania nám sprostredkúvajú dáta, ktoré dostaneme po vykonaní experimentov na danom simulačnom modely. Musíme si uvedomiť, že výsledky experimentov nemusia vždy dopadnúť pozitívne a podľa očakávaní, ale naopak by mali poukázať na správanie modelu v danej situácii pri istých vstupných dátach a za predpokladu, že model je navrhnutý správne a podľa špecifikácie. Aj negatívny výsledok môžeme v konečnom dôsledku považovať za pozitívum, pretože nám odhalil nedostatky zostrojeného modelu. V skutočnosti je vhodné kombinovať simulačné testovanie na vytvorenom modeli a testovanie na reálnom systéme v skutočnosti. (7)

3 Modelovanie a analýza systémov v nástroji UPPAAL

3.1 UPPAAL

Nástroj je výsledkom vývoja univerzít Uppsala a univerzity v Aalborgu. Slúži na overovanie systémov v reálnom čase, ktoré sú reprezentované vo forme modelov. Implementácia bola rozšírená o reprezentáciu premenných typu integer, dátového typu štruktúra a synchronizáciu pomocou kanálov. Prvá verzia nástroju UPPAAL bola vydaná v roku 1995. Od tohto roku sa postupne pracovalo na vývoji a jeho využitie bolo aplikované či už v oblasti komunikačných služieb alebo v oblasti multimédií. V porovnaní s konkurenčnými nástrojmi ako je napríklad SPIN je UPPAAL cenovo dostupnejší, využíva techniky na vyššiu výkonnosť a využíva nové dátové štruktúry. Táto implementácia však nie je dostatočná na zachytenie správania komplexných systémov. Problém je riešený v rámci rozšírenia UPPAAL SMC.(8)

Táto novšia verzia UPPAAL reprezentuje systémy ako sieť automatov, ktorých správanie závisí na stochastických a nelineárnych dynamických znakoch. V rozšírení UPPAAL SMC, je každý komponent systému opísaný modelom, ktorého hodiny môžu byť ohodnotené rôzne.

3.1.1 Využitie

Hlavnou myšlienkou tohto rozšírenia je monitorovanie simulácií daného systému, následné využitie štatistických údajov (získaných testovaním hypotéz alebo simuláciou Monte Carlo) a rozhodnutie či daný model zodpovedá realite. SMC môžeme klasifikovať ako nástroj, ktorého miesto je niekde v strede medzi samotnými testovaním a klasickými technikami na sledovanie modelu. SMC bolo vytvorené ako implementácia menších nástrojov, ktoré boli použité na testovanie v rôznych prípadoch štúdií. Nasledovne bol tento nástroj úspešne aplikovaný v rôznych častiach vývoja a výskumu, v odvetviach ako sú biologické systémy, energetické systémy alebo v softwarovom inžinierstve s nasledovným aplikovaním v priemysle. Hlavnou príčinou úspechu tohto nástroja je, že implementácia, porozumenie a samotné používanie je jednoduché či už pre odborníkov, ktorí s daným nástrojom pracujú, ale aj pre zákazníkov, ktorí nemajú so samotným vývojom a výskumom nič spoločné. Ďalšou výhodou je využitie aj v ďalších odvetviach, odlišných od verifikácie, a to je napríklad plánovanie a robotika.

UPPAAL SMC je použiteľný aj na systémy, ktoré sú nedeterministické (prechody medzi stavmi sú ohodnotené nedefinovaným pravdepodobnostným rozložením), ako napríklad nástroj COSMOS, ktorý je súčasťou SMC a používa sa na hľadanie optimálnych plánovačov pre Markove procesy. V tejto práci sa budeme venovať nástroju UPPAAL SMC, ktorý sa bude zaoberať validovaním vlastností určitého deterministického modelu v danom prostredí. Informácie, ktoré sú použité v celej tejto kapitole boli prevzaté zo zdrojov (9) a (10).

3.1.2 Hlavné časti nástroja

1) Systémový editor (popis modelu)
2) Simulátor

3) Verifikátor

3.1.3 Systémový editor

Priblížime si niektoré špecifické znaky modelovacieho nástroja UPPAAL SMC, ako je grafické rozhranie, simulačný framework a verifikátor. Jedná sa o rozšírenie overovania modelov systémov zo základnej verzie UPPAAL. Pre samotný popis systému sa dajú použiť:

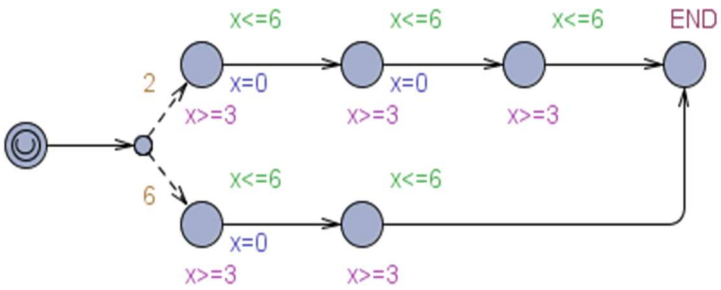
- Globálne deklarácie
- Šablóny procesov
- Definície systému

Systém v UPPAAL je modelovaný ako súbor súbežných procesov. Vlastnosti týchto procesov sú vyjadrené pomocou hodín a dátových premenných. Nad nimi sú definované dátové typy *int, bool a pole integerov* alebo hodnôt typu *bool*. Hodiny a premenné sú súčasťou takzvaných priestorov (ang. scopes), ktoré sú procesom prideľované buď lokálne alebo globálne.

V rámci globálnej deklarácie môžeme zadeklarovať hodiny, konštanty, celočíselné premenné (typu integer), deklaráciu celočíselnej premennej v určitom rozsahu, 2D pole premenných typu integer alebo deklaráciu bitových polí, ktoré budú využívané v tomto systéme. Syntax spomenutých deklarácií je veľmi podobná napríklad jazyku C. Uvedieme si deklaráciu hodín x a y:

```
clock x, y;
```

Súčasťou nedeterministického modelu sú časové oneskorenia, ktoré sú zahladzované pravdepodobnostným rozložením. Na úrovni komponentov je využívané uniformné rozdelenie v prípade ohraničených oneskorení, a v prípade neohraničených oneskorení sa využíva exponenciálne rozdelenie.



Obrázok 3.1: Príklad modelu (reprezentovaný konečným automatom)

Uvažujme model, ktorý sa nachádza na obrázku 3.1. Tento model má 7 stavov. Prvý je počiatočný stav, ktorý je na obrázku znázornený v dvojitom krúžku so znakom podobným písmenu U v strede. Samotný význam tohto znaku v strede začiatočného stavu si vysvetlíme za chvíľu. Každý model by mal obsahovať maximálne jeden počiatočný stav, z ktorého sa simulácia systému začína. Konečný stav automatu je označený pomenovaním END. Na modeli si môžeme všimnúť ohodnotenia jednotlivých stavov a hrán, ktoré vedú z počiatočného stavu a ďalej sa vetvia. Tieto čísla nám udávajú minimálne a maximálne oneskorenie v každom stave tohto automatu. Celkovú dosiahnuteľnosť konečného stavu nám udáva suma jednotlivých uniformných rozdelení v stavoch modelu systému.

Čiže konkrétne v našom prípade by sme mali počítať s dosiahnutím konečného stavu v intervale [6,18]. Podmienku *x<=6* v danom stave nazývame invariantom.

V rámci modelu rozlišujeme:

- Invariant – Mohli by sme ho definovať ako výraz, ktorý je definovaný pomocou hodín, konštánt alebo celočíselného výrazu typu integer. V podstate sa jedná o kombináciu podmienok, ktoré sú zložené z hodín a celočíselných premenných typu integer. Vo všeobecnosti je to obmedzenie času (výhradne zhora), tráveného v stave, ktorému je invariant priradený.
- Stráž[1] - Na úrovni prechodov sú definované takzvané stráže. Na obrázku 3.1 je to podmienka *x>=3* znázornená nad prechodom medzi jednotlivými stavmi. Formálne by sa stráž dala definovať ako určitý výraz, ktorého výsledok je rovný pravdivostnej hodnote. Jedná sa o konjukciu časových a dátových obmedzení. Stráž vieme zapísať ako postupnosť výrazov oddelených čiarkami. Zápis tohto časového výrazu môže byť vyjadrený pomocou konštánt a premenných podľa syntaxe jazyka C. Výsledok podmienky je vyjadrený ako rozdiel hodnoty hodín, ktorý je porovnávaný s celočíselným výrazom. Stráž by sme mohli definovať ako všeobecné obmedzenie vykonania hrany podmienkou, v tomto prípade *x>=3*.
- Aktualizácia[2] - Aktualizáciu, ktorá je na modeli 3.1 vyznačená ako podmienka *x=0* modrou farbou, môžeme definovať ako zoznam priraďovacích príkazov. Každý výraz je vyjadrený ako *x:=e,* kde *x* môže predstavovať premennú alebo hodiny a *e* celočíselný výraz. Tento výraz je vyhodnocovaný zľava doprava.

Model v UPPAAL SMC pozostáva zo siete navzájom komunikujúcich komponentov jednotlivých procesov, ktoré sú súčasťou modelu. Dva procesy môžu v rámci modelu komunikovať pomocou komunikačných kanálov a globálnych premenných.

Kanály pomocou, ktorých procesy komunikujú môžeme rozdeliť na :

- Komunikačný kanál – príkladom môže byť kanál *d*, ktorý by sme syntakticky zapísali ako *chan d;*. Prechody v procesoch kde sa využíva komunikačný kanál sú označované ako *d!* (signál typu akcia, vysielanie) alebo d? (signál typu udalosť, príjem). Vysielač je blokovaný, kým príjemca nie je pripravený na príjem.
- Naliehavý kanál – syntakticky ho môžeme zapísať ako *urgent chan d;*. Tento prípad synchronizácie procesov prebieha bez vzájomných oneskorení. Čiže je to príjem v tom čase, v ktorom sa hrana príjemcu stáva vykonateľnou. Vysielač je blokovaný ak príjemca nie je pripravený na príjem. Zároveň platí pravidlo, že na hranách ohodnotených synchronizačnými kanálmi (*d!,d?*), nie je možné definovať stráže.
- Kanál pre viacsmerové vysielanie – Hovorí sa mu aj tzv. broadcast. Správne ho zapisujeme ako *broadcast chan e;*. Umožňuje synchronizáciu typu jedného ku veľa procesom (1-to-many). Akciu, ktorá popisuje začiatok broadcastu označujeme *e!*. Ktorýkoľvek vykonateľný prechod, ktorý čaká na prechod *e?*, bude synchronizovaný s vysielajúcim. Na hranách tohto typu kanálu nie sú povolené stráže. Zároveň platí pravidlo, že vysielač nie je blokovaný.

V rámci systému môžeme definovať aj tzv. šablónu. Šablóna nám pomáha pri vytváraní nových procesov a sprehľadňuje celkový pohľad na vytvorený systém. Definuje spoločné vlastnosti

[1] Anglicky Guard

[2] Anglicky Update

procesov toho istého typu. Každý vytvorený proces sa potom stáva inštanciou danej šablóny. Šablóna môže obsahovať symbolické premenné a konštanty, ktoré môžeme nazvať aj parametre šablóny, ktoré sa používajú na vytváranie nových inštancií procesov. Syntax je podobná ako pri deklaráciách, ktoré sme si vysvetľovali, ale bez možnosti inicializácie. Súčasťou šablóny môžu byť aj lokálne hodiny a premenné, ktoré sú syntakticky rovnaké ako tie globálne.

Systém je tvorený množinou procesov, ktorý tvoria tento systém. Každý proces v systéme musí byť buď:

- Proces, ktorý sa vyskytuje na ľavej strane priraďovacieho príkazu pre inštanciu procesu
- Šablóna bez parametrov

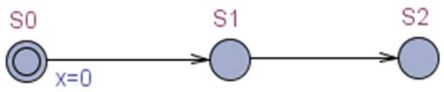
```
P1=P(1);
P2=P(2);

System P1,P2;
```

Jednotlivé typy stavov rozdeľujeme na:

- Počiatočné (tzv. Initial) – Tento stav môžeme definovať ako začiatočný v rámci daného procesu. Označuje sa dvojitým krúžkom. Definovaný môže byť aj spolu s invariantom a aj bez neho.
- Bežné – V tomto prípade sa jedná o obyčajný stav v danom modeli. Tak isto môže byť zadefinovaný buď s invariantom alebo bez neho.

P0

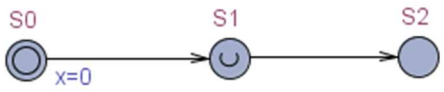


Obrázok 3.2: Bežný stav

Keď je P0 v stave S1, ktorý nazývame bežný, čas plynie a môže byť zvolený ktorýkoľvek prechod.

- Urgent – V tomto type stavu nemôže systém stráviť viac ako 0 časových jednotiek. Inak povedané systém strávi v danom stave 0 časových jednotiek.

P1



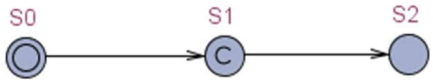
Obrázok 3.3: Stav urgent

V prípade, že P1 sa nachádza v stave S1, ktorý taktiež nazývame aj stav urgent, tak čas v tomto stave neplynie a prechod môže byť zvolený ľubovoľný.

- Committed – Oproti predchádzajúcemu stavu Urgent, tu platí, že zmena derivácie v čase na danom mieste je nulová. Navonok sa tento stav prezentuje ešte prísnejšie. Ak nastane situácia, že systém je v mieste typu F, množina vykonateľných hrán je zúžená na množinu vychádzajúcich hrán zo stavu typu F (tieto prechody sa vykonajú

prednostne pred ostatnými). Tento typ prechodu je stav, ktorým sa dá obmedziť nedeterminizmus vykonávateľných prechodov na množinu prechodov vedúcich z miest F.

P2



Obrázok 3.4: Stav commit

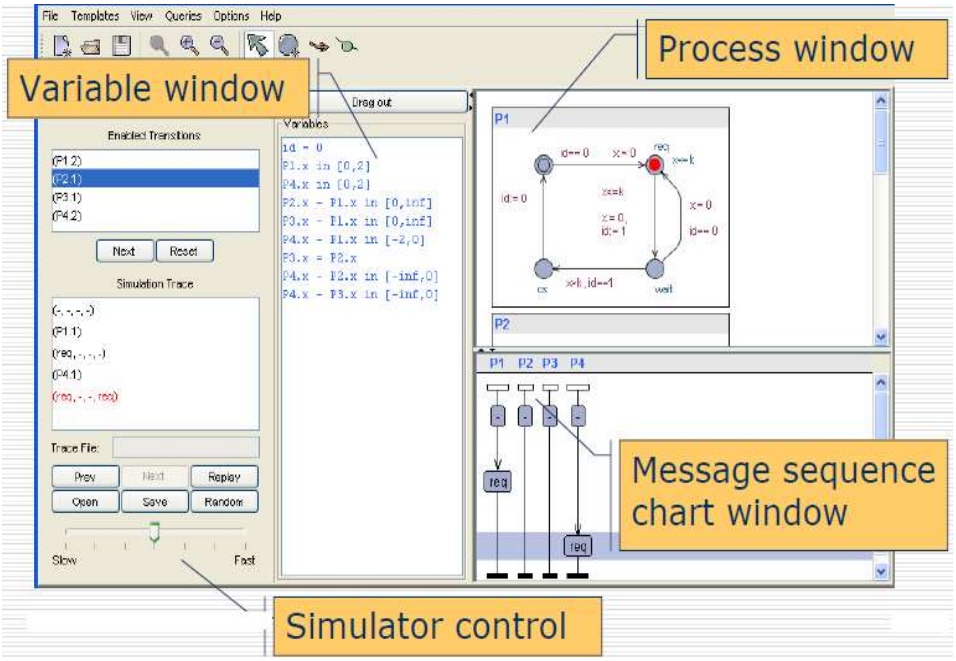
Ak sa P2 nachádza v stave S1, ktorý je na obrázku označený aj ako stav commit, tak týmto označením definujeme, že v danom stave čas neplynie a jediná hrana, ktorá môže byť použitá je hrana z S1 do S2.

3.1.4 Simulátor

Simulátor by sme mohli definovať ako nástroj, ktorý sa stará hlavne o :

- Interakciu systému s užívateľom
- Sledovanie chovania systému
- Vizualizáciu behu systému

Tento validačný nástroj dovoľuje užívateľom pozorovať a reagovať na správanie daného systému v priebehu vykonávania. Takisto sa používa na vizualizáciu vykonávania generovaného verifikátorom.



Obrázok 3.5: Časti simulátora

Simulátor môže byť využitý podľa troch hlavných smerov. V prvom prípade môže užívateľ spustiť systém manuálne a vybrať, s ktorým prechodom sa chce zaoberať. Druhý spôsob je, že

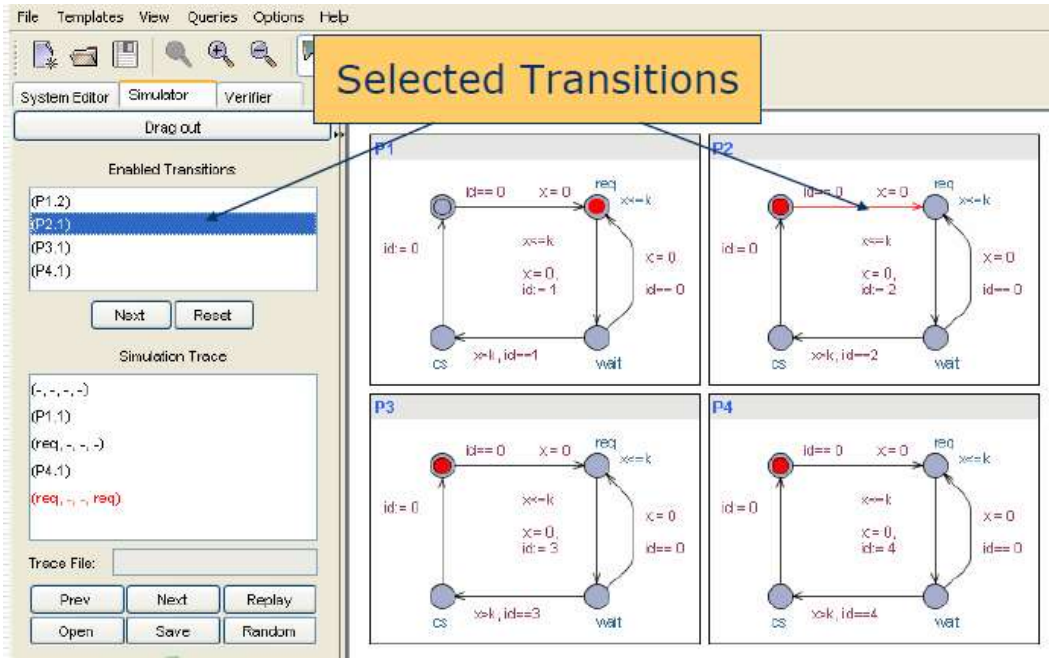
v základnom móde môže byť nastavený tak, že systém beží od začiatku bez zásahu užívateľa. V ďalšom prípade môže užívateľ prechádzať jednotlivé kroky (uložené alebo importované z verifikátora) na zistenie ako sú jednotlivé stavy dosiahnuteľné.

GUI simulátora je zložené zo štyroch hlavných častí. Jednotlivé časti tohto simulátora sú zobrazené na obrázku 3.5. Sú to:

- Procesy (ang. Process window)
- Premenné (ang. Variable window)
- Riadenie simulácie (ang. Simulator control)
- Záznam o činnosti systému (ang. Message sequence chart window)

Okno procesov zobrazuje jednotlivé inštancie procesov v danom systéme.

- Aktuálny stav každého procesu (automatu) je vyfarbený načerveno
- Vykonateľné prechody, zvolené aktuálne v simulačnom okne, sú v príslušných automatoch rovnako červeno zafarbené.



Obrázok 3.6: Priebeh procesu v rámci simulátora

Okno premenných zobrazuje hodnoty premenných a hodín v aktuálnom stave alebo na zvolenom prechode.

Časť simulátora, ktorá zabezpečuje riadenie simulácie dovoľuje užívateľovi riadiť simuláciu a voliť (symbolický) stav alebo prechod, ktorý má byť vizualizovaný. V hornej časti tohto okna 3.6 je popísaná simulácia krok za krokom, ktorá zobrazuje užívateľovi všetky prechody, ktorými je možné prejsť. Naopak spodná časť poukazuje na samotné stopy systému. Stopa je alternatívna postupnosť miest a prechodov.

Záznam o činnosti systému je vyjadrený postupnosťou správ generovaného behu. Zobrazuje správy pomocou diagramu na základe vygenerovaných krokov. V tomto diagrame rozlišujeme dve hrany. Horizontálna hrana, pre každý synchronizačný bod a vertikálna hrana pre každý proces.

3.1.5 Verifikátor

Verifikátor sa používa na overovanie invariantov a vlastností typu:

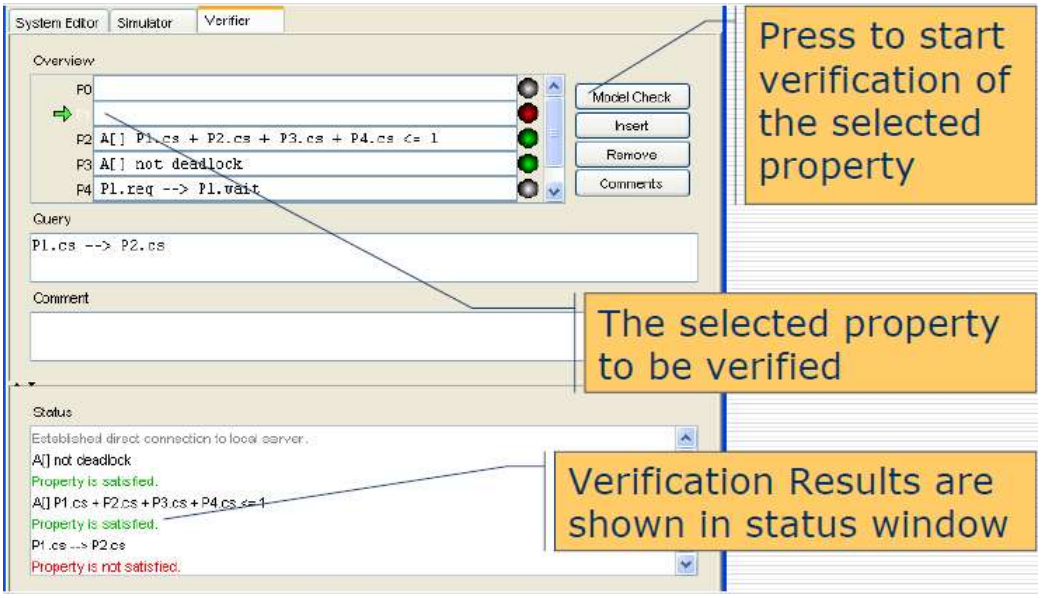
- dosiahnuteľnosť – zisťujeme, či existuje nejaká cesta z počiatočného stavu, ktorou by sme sa dostali do požadovaného stavu. Nezaručuje nám to však korektnosť daného protokolu, ale validuje základné správanie daného modelu. Vhodným príkladom by mohlo byť poslanie správy odosielateľom.
- bezpečnosť – hovorí o tom, že sa nestane nič zlé (napríklad úroveň teploty jadra nukleárnej továrne sa drží pod určitou hranicou, čiže sa nachádza v bezpečnom rozmedzí)
- živosť – v tomto prípade sa jedná o situáciu kde sa následkom niečoho „dobrého“ stane potom niečo ďalšie (napríklad v rámci komunikačného protokolu nastáva prípad, kde správa, ktorá bola odoslaná by mala byť aj prijatá)

UPPAAL využíva množstvo nových dotazov v rámci stochastickej interpretácie časových automatov. Umožňuje zobrazovanie hodnôt výrazov (prepočítaných na hodiny alebo integer) počas simulačného času. Toto udáva celkový náhľad na systém a poskytuje možnosť testovania niektorých zaujímavých vlastností daného modelu. Syntax je nasledovná:

simulate N [<= bound] [E1,..,Ek]

V tomto prípade N predstavuje prirodzené číslo, ktoré predstavuje počet simulácií, ktoré budú spravené. Bound je čas, ktorý je závislý na simuláciách a E1,..,Ek sú výrazy v stavoch, ktoré sú vizualizované a monitorované.

UPPAAL využíva zjednodušenú verziu jazyka CTL ako takzvaný dotazovací jazyk. Tento jazyk sa skladá zo stavových formúl a formúl definovaných v rámci prechodov medzi stavmi, ktoré tvoria cesty. Stavové formuly poukazujú na konkrétne stavy, v ktorých sa model môže nachádzať.



Obrázok 3.7: Verifikátor

Vlastnosti, ktoré pomáhajú k sledovaniu modelu procesu implementovaného v nástroji UPPAAL dodržiavajú sémantiku konečnej množiny stavov. V rámci dotazovacieho jazyka si definujeme nasledujúce pravidlá:

- A [] p – pre všetky vetvenia platí, že *p* je splnené
- E <> p – existuje také vetvenie, že *p* je niekedy splnené
- E [] p – existuje také vetvenie, že *p* je vždy splnené
- A <> p – pre všetky vetvenia platí, že *p* je niekedy splnené
- p → q – kdekoľvek je *p* splnené, *q* môže byť splnené

P je výraz, ktorý nemá žiadny ďalší vedľajší efekt, poprípade sa jedná o stavovú formulu (tj. konjukcia miest).

Na nasledujúcich príkladoch si ukážeme aj zložitejšie konštrukcie dotazov, ktoré nám slúžia na overovanie behu celého systému a môžu nám pomôcť pri hľadaní a opravovaní chýb. Stavy, ktoré sú testované, boli vybrané na základe simulácie príkladu prechodu vlaku cez železničné priecestie. Tento príklad je zobrazený v príručke nástroja UPPAAL.

a) `Pr[<= 300](<> Gate.len < 3 and t > 20)`
   V tomto prípade nám číslo 300 udáva že simulácia bude bežať maximálne dovtedy kým nadobudne hodnotu 300. Výraz, ktorý je v zátvorkách testuje, či nastane situácia, že priecestím prejdú tri vlaky a čas pritom presiahne hodnotu 20.
b) `Pr[<= 100](<> Train(0).Cross) >= 0.2`
   Tu overujeme situáciu, či môže niekedy nastať stav, keď vlak 0 prejde priecestie za 100 alebo menej časových jednotiek. Podmienka >=0.2 nám vraví či je táto pravdepodobnosť väčšia alebo menšia ako 0.2
c) `Pr[<= 100](<> Train(5).Cross) >= Pr[<= 100](<> Train(0).Cross)`
   Týmto dotazom testujeme pravdepodobnosť, že vlak 5 prejde priecestie skôr než vlak 0 za 100 časových jednotiek.
d) `A[] not deadlock`
   Tento typ dotazu poukazuje nato, že v systéme sa deadlock nenachádza.
e) `E[ <= 100; N](max: sum(i:id_t) Train(i).Stop)`
   Rozdiel oproti prípadu v odrážke c) je taký, že simulácia sa nám bude opakovať N krát. Okrem toho bude tento výraz vracať najvyššiu hodnotu, ktorá bude určená z hodnôt vrátených funkciou sum, potom čo každá úloha nadobudne stav *Train(i).Stop.*
f) `inf {Task(2).EndPeriod}: Task(2).wcrt`
   Týmto výrazom testujeme infímum najdlhšej doby vykonávania úlohy v systéme. Naopak nahradením *sup* za *inf* by sme hľadali suprémum.

Verifikátor nástroja je zobrazený na obrázku 3.7. Jednotlivé vlastnosti modelu systému, sa dajú overovať na základe dotazov definovaných v časti *Overview.* Užívateľ môže vytvárať a následne pozorovať správanie jednej alebo viacero vlastností. Do časti *Overwiew* sme schopný pridávať a odoberať ďalšie dotazy, ktorými chceme overovať správanie modelu a tak isto pridávať komentáre ku jednotlivým dotazom v zozname. V prípade výberu dotazu, sme schopný editovať definíciu, poprípade pridať komentár, kvôli ľahšiemu pochopeniu daného dotazu. Tieto komentáre sa používajú hlavne pri zložitých dôkazoch, kde nie sme na prvý pohľad schopný určiť čo daný dôkaz overuje.

Panel *Status* zobrazený v dolnej časti obrázku 3.7 nám zobrazuje priebeh komunikácie so samotným serverom. Model systému rozoznáva či k danej časti simulácie existuje nejaký dôkaz na overenie. Dotazy, ktoré boli vyhodnotené ako vyhovujúce danému správaniu, sú označené zelenou farbou a tie, ktoré nevyhovujú červenou. Môže nastať situácia keď po vyhodnotení bude tento dotaz označený žltou farbou. To znamená, že s danou aproximáciou je verifikácia nepresvedčivá.

V prípade, že sa snažíme spustiť väčšie a zložitejšie verifikačné úlohy, zistíme, že je nepohodlné spúšťať toto overovanie cez GUI. V tomto prípade je jednoduchšie a prehľadnejšie spúšťať dané úlohy pomocou stand-alone príkazového riadku zvaného *verifyta*. Ďalším riešením je spustenie verifikácie zo vzdialeného UNIX počítača. V rámci druhého variantu sú povolené všetky argumenty, ktoré sú validné v prípade normálneho spustenia cez GUI.

4 Plánovanie úloh s obmedzeniami

Pri riešení návrhu systému je dôležité, aby systém reagoval na podnety nie len správnou odozvou, ale tak isto aby ju poskytol včas, a to v predom stanovenom časovom intervale meranom od vzniku podnetu. Často sú tieto systémy označované ako systémy pracujúce v reálnom čase, systémy pracujúce s reálnym časom alebo systémy pre riadenie v reálnom čase (RTS). Aj keď si to možno neuvedomujeme, stretávame sa s nimi v každodennom živote. Vyskytujú sa ako súčasť časovo kritických úloh v mnohých oblastiach ľudskej činnosti, od jednoduchých riadiacich systémov vyskytujúcich sa v bežných domácnostiach (mikrovlnné trúby, pračky, chladničky), cez systémy zabudované v komunikačných a multimediálnych zariadeniach (mobilné telefóny, digitálne fotoaparáty, kamery, prehrávače DVD, herné konzoly), dopravných prostriedkoch (automobily, lietadlá), lekárskych prístrojoch (prístroje na monitorovanie životných funkcií človeka), až po komplexné systémy riadiace priemyselné či armádne zariadenia.(11)

Požiadavky na tieto systémy sa môžu výrazne líšiť, a to nie len z hľadiska časovej kritiky, ale taktiež z hľadiska bezpečnosti a vykonateľnosti. Taktiež si musíme uvedomiť význam slova včas, ktorého význam závisí na konkrétnej úlohe, takže rovnosť odozva v reálnom čase a okamžitá odozva neplatia. Odozva kladená skôr môže byť vo výsledku taká istá ako žiadna odozva, respektíve zlá odozva. Platí však, že čím vážnejšie následky môže mať nedodržanie časových medzí kladených na jednotlivé odozvy, tým väčšie požiadavky sú kladené na daný systém. Ako výsledok nedodržania časových medzí môžeme uviesť bezhotovostnú transakciu platobnou kartou alebo systém prejazdu cez železničné priecestie. V prvom prípade bude následkom nedodržania medzí (spôsobené napríklad výpadkom serverov na strane banky) k predĺženiu celkovej odozvy alebo v horšom prípade ku nevykonaniu platobnej transakcie. Takéto správanie môže viesť ku opakovaniu celej transakcie alebo ku zaplateniu v hotovosti. Táto situácia môže v prípade, že zákazník nemá hotovosť viesť k nepríjemnej situácii, ktorá výrazne znižuje kvalitu služieb. Ale tieto následky nedodržania časových medzí sa nedajú považovať za vážne. V prípade železničného priecestia môže nedodržanie medzí spôsobiť veľmi vážne problémy. V tomto prípade môže systém reagovať v nesprávnom čase na príjazd vlaku k prejazdu alebo odjazd vlaku z prejazdu neskorším či predčasným pohybom závor, aktiváciou zvukového alebo svetelného zariadenia. Jednotlivé časové odozvy je nutné z hľadiska časovej náročnosti rozlišovať. Preto odozvy delíme na:

- Mäkké
- Tvrdé
- Pevné

Mäkká odozva je optimálna odozva pre získanie užívateľom očakávanej kvality služieb poskytovanej systémom. Nedodržanie medzí odozvy, napríklad v dôsledku preťaženia systému nadmerným počtom podnetov, môže spôsobiť dočasné zníženie kvality služieb so zanedbateľným vplyvom na okolie. Ako príklad by sme mohli uviesť stlačenie tlačidla výťahu.(11)

Tvrdá odozva je taká kde musí platiť, že každá odozva tohto typu musí byť bezpodmienečne dodržaná. Nedodržanie jedinej z nich vedie k trvalým následkom, ktoré nie sú z celkového hľadiska vratné. Každé ďalšie pokusy o dodržanie medzí sú nemožné alebo zbytočné, pretože vplyvom zlyhania podstatnej časti systému už neexistuje spôsob vedúci k zmierneniu už vzniknutých následkov. Ako príklad by sme mohli uviesť časové rozmedzie pre zasunutie regulačných tyči do jadrového reaktoru po zistení poruchy v chladiacom okruhu.(11)

Pevná odozva je taká, kde sa odozva tohto typu vyznačuje dopredu zadanou toleranciou nedodržania. Prekročenie tejto tolerancie môže mať v konečnom dôsledku vážny dopad na okolitý

systém. V prípade, že je tolerancia navrhovaná s dostatočnou rezervou, systém môže predvídať jej potencionálne prekročenie s predstihom dostačujúcim ku včasnému spusteniu zotavovacích mechanizmov. V prípade prekročenia času tolerancie a neschopnosti zotavenia sa z daného stavu, je možné riešiť vzniknutú situáciu prostriedkami mimo systém. Následky nedodržania medzí sú typicky vratné a môžu byť dodatočne zmiernené alebo až odstránené za podmienky, že nezlyhajú záchranné mechanizmy. Ako príklad sa dá brať časové rozmedzie pri ventilácii pľúc pacienta na jednotke intenzívnej starostlivosti, kde je pri prekročení tolerancie 1s posielaný signál doktorovi.(2)

Podľa danej špecifikácie je možné konštruovať aj systémy RT. Typ systému je určený na základe časovo najkritickejšej medze, ktorá je prezentovaná vstupnými požiadavkami na daný systém. Zvlášť tvrdé systémy musia byť konštruované tak, aby už spôsobom ich návrhu bolo možné vylúčiť, že budú prekročené niektoré medze odoziev pri splnení všetkých požiadaviek daných špecifikáciou systému.

Táto špecifikácia systému je často zapisovaná prostredníctvom prostriedkov prirodzeného jazyka. Tento spôsob je vyhovujúci hlavne pre zadávateľa systému, pre ktorého je tento zápis najzrozumiteľnejší. Avšak na druhej strane je táto cesta zápisu nevhodná, pretože dochádza k veľkému množstvu nepresností a nedorozumení, hlavne čo sa týka detailov pri špecifikácii systému. Riešením tohto problému je zavedenie formálnej špecifikácie, často zapisovanej prostriedkami logiky reálneho času (RTL), časovaných automatov (TA), nástroja SMV alebo potom pomocou grafického hierarchického popisu systému, ako sú napríklad nástroje Modechart alebo RT-Lotos. Okrem toho, že špecifikácia popisuje chovanie systému RT, je možné ju chápať ako súhrn predpokladov, ktoré musia byť splnené pre zaistenie funkcie daného systému. Pritom je nutné počítať s tým, že pri ich nedodržaní je potrebné očakávať, že chovanie systému môže vybočiť z popisu daného špecifikáciou.(6)

Ak sa požadované vlastnosti systému dajú odvodiť od špecifikácie, to znamená, že medzi požadovanými vlastnosťami a špecifikáciami neexistuje žiadny rozpor, je možné pristúpiť k ďalšej etape vývojového cyklu systému RT a tou je prevod formálnej špecifikácie systému na tzv. množinu úloh RT.

Každú úlohu si vieme predstaviť ako výpočtovú jednotku vykonávajúcu určitú funkciu, ako je napríklad čakanie na vstupný podnet, vzorkovanie, transformácia veličín, meranie času, interakciu s okolím, či generovanie reakcii. Inak povedané, úloha je zodpovedná za správne vykonávanie konkrétnej časti algoritmu vykonávaného v reálnom čase. Podľa toho, ako je možné charakterizovať intervaly medzí príchodmi požiadaviek na zahájenie úloh, tak rozdeľujeme jednotlivé úlohy do týchto kategórií:

- Periodická – požiadavky prichádzajú s periódou T
- Aperiodická – intervaly rozmedzia príchodov požiadaviek sa nedajú nijak ohraničiť
- Sporadická – je známa najkratšia možná doba medzi príchodmi požiadaviek

Prevodom systému na množinu úloh sa priblížime k celkovej realizácii systému, ale musíme brať ohľad nato, že reprezentácia systému abstrahuje jak prostriedky cieľového operačného systému, tak aj vlastnosti cieľovej platformy. Ale niektoré parametre nie je možné zanedbať a je nutné ich zahrnúť do modelu úlohy RT. Medzi takéto základné parametre patrí absolútny čas príchodu požiadavky na vykonanie úlohy (čas vyvolania úlohy), perióda medzi príchodmi požiadaviek na vykonanie úlohy (iba pre periodické úlohy), najdlhšia z možných dôb behu úlohy, relatívne časové rozmedzie odozvy úlohy, absolútne časové rozmedzie odozvy úlohy, či priorita úlohy.(3) Každú úlohu RT obvykle označujeme písmenom gréckej abecedy τ a množinu úloh písmenom Γ. Tak isto môžeme povedať, že čím viacej parametrov model úlohy obsahuje, tým presnejšie zodpovedá reálnemu systému. Tým však rastú nároky na konštrukciu plánovača a analýzu systému RT.

4.1 Plán

Usporiadanie vykonávania úloh v čase nazývame plán. Obvykle je zobrazovaný pomocou časového diagramu. Platia preň nasledujúce tvrdenia, ktoré sme prevzali zo zdroja (12) :

- Je označovaný ako prípustný práve vtedy, ak sú dodržané časové medze všetkých úloh
- Množina úloh je plánovateľná, ak sa pre ňu dá nájsť časový plán
- Mechanizmus generujúci plán je optimálny na množine úloh, keď je schopný pre ľubovoľnú podmnožinu nájsť prípustný plán
- Testom plánovateľnosti nazývame postup, ktorým zisťujeme, či je daná množina úloh plánovateľná.
- Postup umožňujúci rozhodnutie, či pridaním úlohy vzniknutej na základe novej požiadavky do existujúcej množiny úloh zostane táto množina plánovateľná, nazývame test prijatia

Plán je generovaný plánovačom, ktorý rozhoduje o tom, ktorej úlohe bude v danom čase procesor pridelený. Voliť poradie vykonávania úloh tak, aby každá z nich bola dokončená do uplynutia jej časovej medze, patrí medzi základné funkcie plánovača úloh. Väčšina z nich je prioritná, čo znamená, že o tom, ktorej úlohe bude procesor pridelený rozhoduje priorita pridelená každej úlohe. Plánovač garantuje, že vždy beží úloha s najväčšou prioritou. Rozlišujeme plánovače:

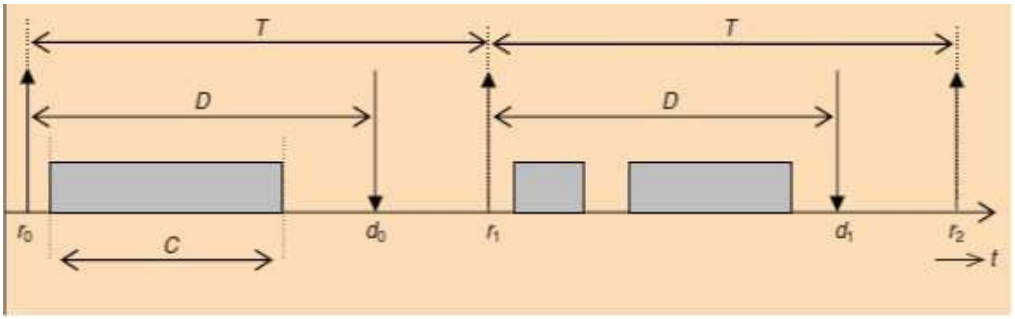
- Preemptívne
- Nepreemptívne

Preemptívny plánovač zaistí prepnutie na najvýznamnejšiu úlohu z momentálne bežiacej úlohy zatial čo nepreemptívny toto prepnutie odloži až do ukončenia momentálne bežiacej úlohy. Doba odozvy preemptívnych plánovačov je kratšia a preto sú v RTOS používané častejšie. V niektorých prípadoch je vhodné tieto dve varianty kombinovať. K hlavným nedostatkom preemptívneho plánovača patrí réžia spojená s prepínaním kontextu každej z úloh. Priamo úmerne rastie s počtom úloh a prepnutí v rámci daného časového intervalu. Túto réžiu tvoria oneskorenie prerušenia, odozva prerušenia a zotavenie sa z prerušenia, ktoré spolu s dobou vykonávania príslušných obslúh prerušení hrajú dôležitú rolu pri určovaní najdlhšej doby odozvy úloh. Navyše môže byť počas obsluhy úlohy zaradená do systému úloha s vyššou prioritou, a v tomto prípade sa doba odozvy prerušenej úlohy výrazne zvyšuje.(12)

Plánovače používajú spôsoby plánovania, ktoré delíme podľa zdroja (12) na:

- Centralizované plánovanie – je implementované a prebieha výhradne na jednej centralizovanej architektúre alebo uzle.
- Distribuované plánovanie – je rozdelené na niekoľko vzájomne komunikujúcich uzlov, z ktorých sa každý podieľa na tvorbe lokálneho plánu podľa predchádzajúcej dohody na stratégii tvorby globálneho plánu (napríklad niektoré úlohy môžu byť priradené konkrétnemu uzlu a neskôr inému uzlu.

Na obrázku 4.1 môžeme vidieť plán s úlohami. Každá úloha je charakterizovaná svojimi parametrami. Parameter *T* určuje periódu úlohy, *D* určuje relatívne časové rozmedzie odozvy úlohy a hodnota *C* udáva maximálnu dĺžku vykonávania danej úlohy. Parameter *r* označuje absolútny čas príchodu požiadavku na vykonanie úlohy a parameter *d* absolútne časové rozmedzie odozvy úlohy.

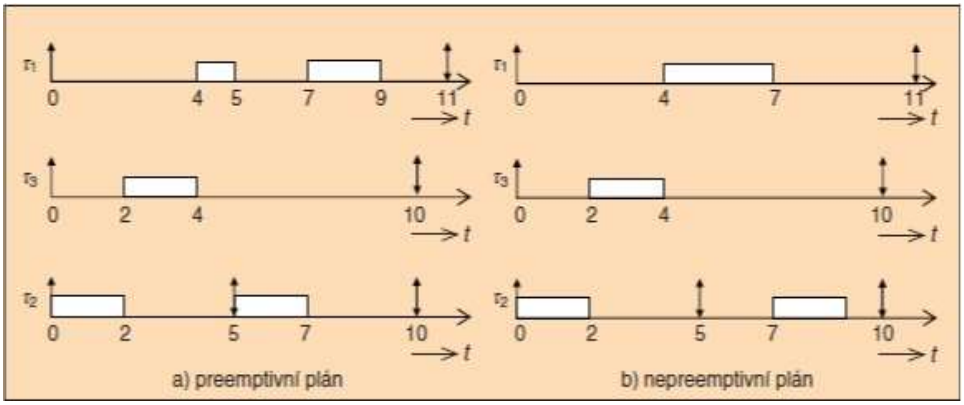


Obrázok 4.1: Plán úloh s parametrami

4.2 Mechanizmy priraďovania priorít

Spôsob akým sú jednotlivým úlohám v časovo kritických systémoch priraďované priority má podstatný vplyv na poradie vykonávania týchto úloh, a teda aj na poradie doby poskytnutia odozvy. Ako sme už spomínali vyššie, väčšina plánovačov používaných v moderných operačných systémoch reálneho času (RTOS) je prioritných a preemptívnych, čomu nasvedčujú hlavne dva dôsledky.(14)

- Vďaka princípu priradďovania priorít je zabezpečené, že procesor je priradený vždy úlohe s najväčšou prioritou
- Preemptivita zabezpečuje minimálnu odozvu obsluhy požiadavky s vysokou prioritou. Takže ak sa momentálne bežiaca úloha nachádza na menej významnej prioritnej úrovni a príde úloha s vyššou prioritou, procesor je jej predaný. Prerušenej úlohe je procesor vrátený po dokončení úlohy s vyššou prioritou, ak do systému nevstúpi úloha s ešte väčšou prioritou.



Obrázok 4.2: Rozdiel medzi preemptívnym a nepreemptívnym plánovaním

4.2.1 Kategórie priraďovania priorít

Podľa toho, či plány produkované mechanizmami priraďovania priorít sú generované pred dobou behu systému alebo až behom nej, delíme mechanizmy na:

- Offline – garantujú optimálnosť plánu
- Online – schopné adaptácie na zmenu v okolí systému

Podľa toho či umožňujú zmenu priorít za behu systému rozdeľujeme mechanizmy na:

- Mechanizmy dynamického priraďovania priorít
- Mechanizmy statického priraďovania priorít

Najjednoduchšie z mechanizmov sú konštruované pre nezávislé periodické úlohy, zložitejšie mechanizmy dokážu priradiť priority i úlohám aperiodickým či sporadickým, závislým úlohám, úlohám plánovaným pri preťažení systému, či v systéme snažiacom sa zotaviť z poruchy. Zatiaľ sa zameriame na preemptívne mechanizmy online pre plánovanie množín nezávislých periodických úloh pre jednoprocesorové prostredie. Z mechanizmov statického priradďovania priorít si predstavíme mechanizmy známe ako RM a DM. V nasledovných podkapitolách sme informácie na vysvetlenie a opísanie mechanizmov, ktoré boli použité, zobrali zo zdroja. (13)

4.2.2 RM (RMA)

V prípade tohto mechanizmu platí, že čím častejšie je úloha volaná, tým vyššia priorita je jej priradená. Významnosť priority úlohy je nepriamo úmerná veľkosti periódy medzi príchodmi jej inštancií, čiže hodnoty statického parametra periódy úlohy. Avšak aj tu môže dochádzať k problémom, ako je napríklad to, že úlohe je pridelený procesor až potom, ako je prekročené časové rozmedzie, ktoré je určené parametrom úlohy. Ak mechanizmus nie je schopný vyriešiť plánovateľnosť množiny úloh RT ako je to v spomenutom prípade, potom existujú dva spôsoby ako túto situáciu vyriešiť. Prvý spôsob je založený na zmene parametrov úloh RT s cieľom zaistiť plánovateľnosť danej množiny úloh RT. Tento spôsob však nie je prijateľný, pretože modifikáciou parametrov úloh RT je možné spôsobiť odklon od už verifikovanej špecifikácie systému RT, a tým aj jeho neočakávané správanie. Preto sa tento spôsob nedoporučuje používať. Druhým spôsobom je použitie iného mechanizmu priraďovania priorít. Ako dôvod zlyhania mechanizmu RM môže byť skutočnosť, že mechanizmus RM je optimálny iba na množine úloh, medzi ktorého parametrami platí vzťah, kde perióda medzi príchodmi požiadaviek sa rovná relatívnemu časovému rozmedziu odozvy úlohy za podmienok, že sa priorita v čase nemení. Inými slovami ak existuje pre množinu splňujúcu podmienky prípustný plán, je možné ho generovať aj pomocou mechanizmu RM. (4)

4.2.3 DM (DMA)

Vychádza z predpokladu, že úlohy s menšou hodnotou časovej medze sú významnejšie. Významnosť priority úlohy je teda v tomto prípade nepriamo úmerná hodnote statického parametru relatívneho časového rozmedzia odozvy úlohy. V porovnaní s mechanizmom RM je mechanizmus DM optimálny na množine úloh, kde je relatívne časové rozmedzie odozvy úlohy menšie ako perióda. Medzi týmito parametrami platí, že relatívne časové rozmedzie odozvy úlohy je podmnožinou periódy úlohy. Pri použití mechanizmu DM sa dajú naplánovať aj tie množiny úloh, ktoré nie sú plánovateľné pomocou mechanizmu RM. Veľmi dôležitú úlohu zohráva existencia testov plánovateľnosti, ktoré tvoria už v etape reprezentácie systému RT množinu úloh RT. Vznikajú skôr ako je samotný systém spustený v reálnom prostredí. Sú schopné rozhodnúť či je daná množina úloh RT so zvoleným mechanizmom priraďovania priorít plánovateľná alebo nie. Rozhodovacia schopnosť týchto testov klesá s počtom parametrov obsadených v modeli úloh RT a so zložitosťou mechanizmu priraďovania priorít úlohám. V niektorých prípadoch však môže nastať situácia, že o plánovateľnosti na základe testov nie je možné rozhodnúť .

Okrem základných mechanizmov, ktoré sme opísali v odsekoch vyššie, existujú mechanizmy, ktoré využívajú dynamické priraďovanie priorít namiesto statického a zaraďujú sa do množiny zložitejších. Tieto mechanizmy dokážu lepšie vyhodnotiť aktuálny stav systému a hlavne rozoznať

blížiace sa prekročenie časového rozmedzia. Hlavný rozdiel oproti mechanizmom RM a DM je, že priority všetkých úloh nachádzajúcich sa v systéme sú premenné v čase, pričom priority sú obvykle prehodnocované v časoch príchodu úloh, dokončenia úloh či prepnutí kontextu úloh. My si predstavíme asi najznámejší z nich, a to je mechanizmus EDF.

4.2.4 EDF

V tomto prípade mechanizmus garantuje, že významnejšia priorita je v čase t pridelená tým úlohám, ktorým v čase t zostáva do uplynutia časového rozmedzia menej času na vykonanie. Táto hodnota je reprezentovaná dynamickým parametrom D(t) úlohy. Mechanizmus funguje na princípe, kde procesor je pridelený úlohe, ktorej deadline je naplánovaný čo najskôr. Ak do systému príde úloha, ktorej deadline je naplánovaný skôr ako je deadline aktuálne bežiacej úlohy, procesor je jej odobraný a je pridelený úlohe, ktorá prišla. Optimálny algoritmus sme schopný pomocou mechanizmu EDF nájsť, pokiaľ existuje prípustný plán testu. Vzhľadom na to, že algoritmus EDF priraďuje prioritu úlohám dynamicky, je pomocou neho možné plánovať periodické aj aperiodické úlohy.

4.2.5 FIFO

Tento algoritmus zvaný aj algoritmus *first in first out* má veľa pozitívnych stránok. Jednou z nich je aj to, že je pomerne jednoduchý na pochopenie a nasledovnú implementáciu. Procesor je prideľovaný úlohám nepreemptívne, podľa poradia príchodu ich požiadaviek, čo je nevýhodné, pokiaľ sa pred úlohami s kratšou dobou vykonávania vyskytuje úloha s dlhšou dobou vykonávania. (1)

4.2.6 Round Robin

Pri tejto stratégii, ktorá sa často označuje skratkou RR, je každej úlohe pridelený procesor najdlhšie na dobu takzvaného časového kvanta[3]. To môže byť konštantné alebo premenné v čase. Ak úloha skončí pred uplynutím tohto kvanta, je procesor pridelený ďalšej úlohe v poradí, inak je úloha prerušená. To znamená, že jej odobraný procesor je pridelený ďalšej úlohe v poradí a prerušená úloha je zaradená za poslednú pripravenú úlohu. Výkonnosť tejto stratégie podstatne závisí na stanovení veľkosti časového kvanta. Väčšie kvantum vedie k dlhším dobám odozvy, zatiaľ čo malé časové kvantum vedie k častejšiemu prepínaniu úloh, ktoré v danom časovom rámci zvyšuje podiel réžie spojenej s prepínaním úloh na úkor vlastnej činnosti úloh.(1)

4.2.7 LLF

Okrem mechanizmu EDF, existujú mechanizmy, ktoré taktiež využívajú dynamické priraďovanie priorít a majú lepšie vlastnosti. Jedným z nich je mechanizmus LLF[4]. Oproti EDF je rozdielny tým, že dokáže zamedziť prekročeniu časových rozmedzí skôr, a to tak, že priority priraďuje úlohám nepriamo úmerne dobe voľnosti úlohy v čase *t*, tj. hodnote parametru *L(t)* úlohy (udávajúceho na ako dlho môže byť úloha odložená bez toho aby prekročila časové rozmedzie). Výsledok tohto vylepšenia sa odzrkadlil na počte prepnutí kontextu a preempcií, ktorý je výrazne väčší ako bol pri mechanizme EDF. Všeobecne platí, že množina plánov generovaných pomocou

[3] Anglicky time slice

[4] Anglicky Least Laxity First

mechanizmu EDF, je podmnožinou množiny plánov generovaných mechanizmom LLF. Ale použiteľnosť mechanizmu je závislá aj od toho, s akou veľkou presnosťou je plánovač schopný určiť zvyšnú dobu behu úlohy. Samotné vyhodnotenie môže byť náročné, pretože hodnotenie nie je možné bez detailnej znalosti cieľovej platformy a príslušného RTOS.

4.3 Priraďovanie priorít závislým úlohám

Doteraz sme predpokladali, že úlohy sú periodické, nezávislé a sú bezchybne vykonávané v rámci jednoprocesorového systému, ktorý je vždy prevádzkyschopný a má dostatočný výkon k včasnému vykonaniu všetkých úloh zahrnutých v systéme. Teraz sa budeme venovať situáciám, kde dané predpoklady nemusia byť splnené. Priority budú priraďované závislým úlohám RT, tj. úlohám medzi ktorými existujú závislosti, ktoré vyplývajú zo špecifikácie systému. Oproti nezávislým úlohám jepotrebné dodržať nielen časové obmedzenia kladené na odozvy úloh, ale naviac pri snahách o ich dodržanie i reflektovať dané závislosti a riešiť problémy z toho vyplývajúce. Rozlišujeme dva typy závislostí:

- Časovú
- Priestorovú

Časová závislosť medzi úlohami existuje vtedy, keď vyplýva zo špecifikácie požiadaviek prednostného vykonania jednej z úloh pred inými. Priestorová závislosť medzi úlohami vzniká, keď viacero úloh v tom istom čase vyžaduje prístup k tomu istému zariadeniu. Je potrebné zaistiť, aby k danému zariadeniu mala prístup výlučne jedna úloha. Tieto závislosti sa môžu vzájomne kombinovať. Práve s týmto typom závislostí je spojené veľké množstvo problémov, ktoré je treba riešiť. V nasledujúcej časti si skúsime predstaviť prehľad tých najvýznamnejších.

4.3.1 Hladovanie

Hladovanie úloh nepredstavuje na triede systémov RT závažný problém. Práve naopak, patrí k základným vlastnostiam systémov RT v tom zmysle, že s klesajúcou významnosťou priority, doba čakania úloh na pridelenie procesoru rastie. Toto čakanie je prirodzené a plynie zo samotnej podstaty systému RT, pre ktorý je kľúčové dodržať časové obmedzenia aj za cenu hladovania niektorých úloh. Avšak ostatné problémy, ktoré spomenieme, vyžadujú viacero pozornosti, pretože ich následkom môže byť oneskorenie odoziev úloh a následne to vedie k nedodržaniu časových obmedzení. Preto je potrebné zvýšiť pozornosť pri vypracovaní návrhu a realizácii systému RT.

4.3.2 Blokovanie

Pri blokovaní už hovoríme o probléme s dopadom na dodržiavanie časových obmedzení. Jedná sa o čakanie úlohy s vysokou prioritou na uvoľnenie prostriedkov momentálne užívaných úlohou s nízkou prioritou. Toto čakanie je však nežiadúce, pretože prioritná úloha sa chystala prednostne vykonať ďalšiu časť svojho kódu, ale vplyvom okolností musela byť pozastavená až do času uvoľnenia tohto zdroja, čo v konečnom dôsledku predlžuje dobu vykonávania a odozvy prioritnej úlohy. Je dôležité spomenúť, že k prostriedkom so vzájomne výlučný prístupom sa obvykle pristupuje v rámci tzv. kritických sekcií (KS). Ich vykonávanie je atomické, a to znamená, že v KS sa môže v jednom momente nachádzať najviac jedna úloha. Kritická sekcia môže byť voľná alebo obsadená prostriedkom R.

4.3.3 Inverzia priorít

Pri plánovaní preemptívnych úloh s pevnými prioritami a riadením prístupu ku kritickým prostriedkom pomocou mechanizmu kritických sekcií môže prísť k inverzii priorít. V tomto prípade sa jedná o obsadenie prostriedku úlohou, ktorá jednak nemusí zdieľať prostriedky s úlohou, ktorá je zablokovaná ale ani nemá vyššiu prioritu. No napriek tomu je spustená skôr, a tým sa podieľa na náraste doby odozvy blokovanej úlohy. Táto situácia je v rozpore s definíciou plánovania a môže viesť k prekročeniu časových rozmedzí úloh. Pokiaľ nie je použitý špecifický mechanizmus, tak doba odozvy úloh nemôže byť v tomto prípade nijako ohraničená, a ani nemôže byť garantované jej rozmedzie.

4.3.4 Uviaznutie

Ako posledný z problémov súvisiacich so zdieľaním prostriedkov spomenieme uviaznutie, tzv. deadlock. Na rozdiel od predchádzajúcich problémov môže uviaznutie viesť k celkovému zastaveniu systému, a preto je dôležité mu predchádzať. K uviaznutiu dochádza vtedy, keď úlohy zdieľajú aspoň dva kritické prostriedky. V podstate nastáva situácia, kde dva alebo viac procesov čaká neobmedzene dlho na podmienky, ktoré reálne nemôžu nastať. Typicky je to napríklad situácia dvoch procesov, ktoré si vzájomne blokujú prístup do kritických sekcií. Pokiaľ nedôjde k uvoľneniu jednej kritickej sekcie, do ktorej jedna z úloh chce vstúpiť, tak ani jedna úloha neuvoľní kritickú sekciu, ktorú zamkla. Táto situácia vedúca k uviaznutiu dvoch úloh môže byť rozšírená na viacero úloh. Uviaznutie predstavuje vážny problém pre kritické RT aplikácie, preto existujú postupy ako vzniku uviaznutia zabrániť.

5 Modely úloh a plánovačov v UPPAAL SMC

V tejto kapitole sa budeme venovať popísaniu a vysvetleniu modelov jednotlivých úloh a plánovačov, ktoré boli vytvorené na základe jednotlivých mechanizmov, opísaných v predošlej časti tejto práce. Postupne si popíšeme stavy a prechody medzi stavmi na jednotlivých schémach modelov mechanizmov.

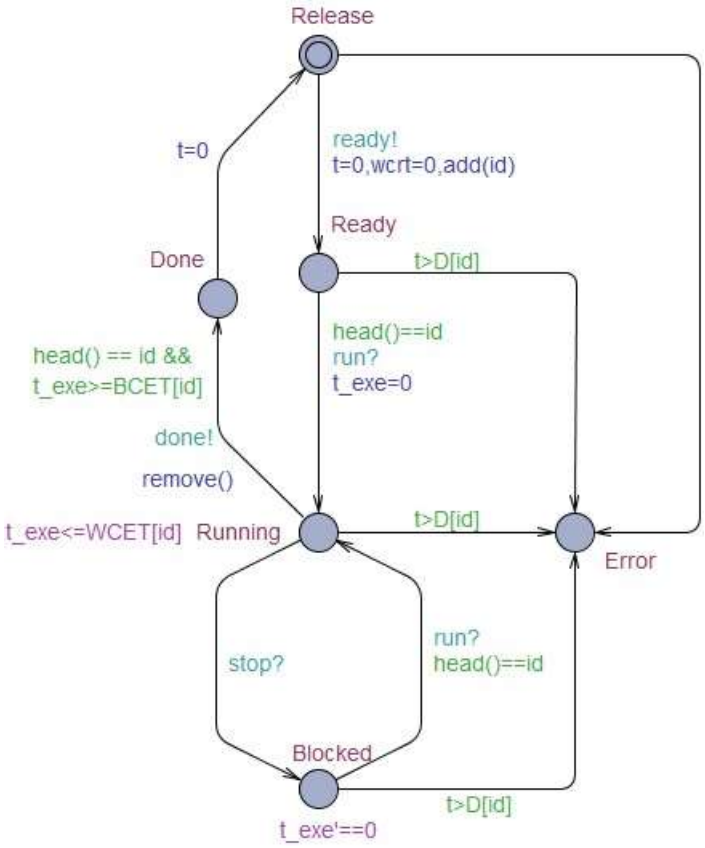
5.1 Základný model úloh

Najskôr by sme stavy a prechody medzi stavmi, ktoré úlohy nadobúdajú počas behu systému popísali na vzorovom modeli. Tento model tvorí zväčša základnú časť pri každom modeli, ktoré si v ďalších podkapitolách vysvetlíme. Čitateľov je ale potrebné upozorniť, že nasledujúci model je len vzorový a použijeme ho len na vysvetlenie základnej konštrukcie, nejedná sa o funkčný model.

Na obrázku 5.1 je zobrazený model, ktorý tvorí kostru modelov pre jednotlivé úlohy. Na zobrazenom vzore môžeme vidieť šesť stavov, do ktorých sa úloha počas behu môže dostať. Sú to stavy *Release, Ready, Running, Done, Error a Blocked*. Do stavu *Release* sú zaradené postupne všetky úlohy, ktoré sú užívateľom zadefinované v deklaračnej časti modelu systému. Do tohto stavu sa úloha dostane v prípade, že je splnený invariant v počiatočnom stave. Je dôležité spomenúť, že aperiodická a sporadická úloha môžu v stave *Release* zotrvať ľubovoľne dlhú dobu, zatiaľ čo periodická úloha ho musí opustiť do vypršania ohraničenia, ktoré je definované jittrom spustenia tejto úlohy. Na tomto mieste čakajú úlohy, pokiaľ nie je splnená podmienka tzv. *Stráž* na prechode do stavu *Ready*. Táto podmienka je definovaná podľa typu mechanizmu, ktorým je úloha plánovaná, a tak isto záleží aj na tom či je úloha periodická, sporadická alebo aperiodická. Na tomto prechode dochádza k aktualizácii, kde sa volá funkcia *add(id)*, ktorá pridáva úlohu do fronty úloh, prípadne sa chová podľa špecifikácie daného mechanizmu a dodatočne dochádza k preusporiadaniu úloh v rámci fronty. Pri rozličných mechanizmoch plánovania sa táto funkcia môže líšiť. V stave *Running* sa nachádza aktuálne bežiaca úloha, pokiaľ nedokončí svoje vykonávanie. Tu je definovaná aj stráž, kde musí byť splnená podmienka, že čas vykonávania musí byť menší ako najhoršia doba vykonávania definovaná pri danej úlohe. Zároveň je úloha odstránená z fronty úloh určených na vykonávanie pomocou funkcie *Remove()*.

```
void remove()
{
  int i;
  for(i = 0; i+1 < N; ++i) { queue[i] = queue[i+1]; }
  queue[--len] = 0;
}
```

Aktuálne vykonávanú úlohu nám vracia funkcia *head(),* ktorá je rovnako definovaná v každom modeli, ktorým sa neskôr budeme zaoberať. Zároveň je na prechode definovaný invariant, ktorým overujeme, že doba vykonávania je väčšia ako najlepšia doba vykonávania danej úlohy. Následne sa úloha dostáva do stavu *Done.* Predtým ako sa úloha vráti späť do stavu *Release* dochádza ku aktualizovaniu hodín *t* na hodnotu 0.



Obrázok 5.1: Vzorový model úloh

Model komunikuje a je synchronizovaný s modelom plánovača pomocou synchronizačných kanálov[5]. Tieto kanály sú v deklaračnej časti definované nasledovne.

```
broadcast chan   done, ready, run, stop;
```

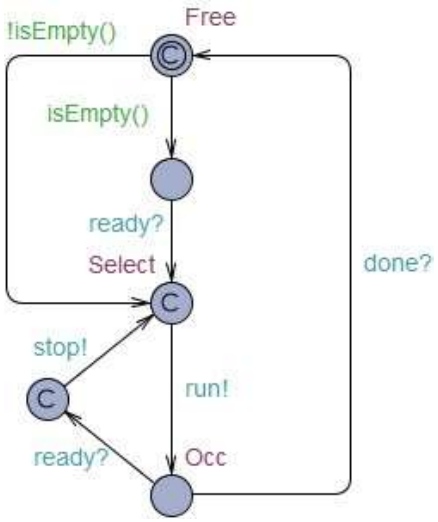
Ako ďalší si predstavíme stav *Error*, do ktorého sa úloha môže dostať zo stavu *Release, Ready, Blocked* a *Running.* Do tohto stavu sa úloha dostane v prípade, že čas, ktorý úloha strávila v systéme prekročil *deadline*[6]*,* definovaný pre danú úlohu.

Posledný, ktorý si opíšeme je stav *Blocked.* Do tohto stavu sa úloha dostane v prípade, že inej úlohe bola pridelená väčšia priorita než úlohe, ktorá bola v danom momente vykonávaná. V takejto situácii je úloha presunutá do stavu *Blocked.* Tento presun je zabezpečený synchronizačným kanálom, ktorý je na strane modelu vyvolaný definíciou *stop?.* Pozastavená úloha zostáva na tomto mieste pokiaľ úloha, ktorá mala prioritu neprejde do stavu *Done.* Moment presunu tejto úlohy späť do *Running* je vyvolaný synchronizačným kanálom *run?.* Po návrate do stavu *Running* úloha pokračuje vo svojom vykonávaní tak ako za normálnych okolností, pokiaľ opakovane nedôjde k uprednostneniu inej úlohy z fronty. Stav *Blocked* využívame v každom modeli, s ktorým budeme pracovať, okrem modelu s mechanizmom plánovania FIFO. Dôvod si vysvetlíme v podkapitole určenej práve tomuto mechanizmu.

Model plánovača sa v prípadoch mechanizmov, ktorým sa budeme venovať bude líšiť len minimálne alebo sa bude zhodovať so vzorovým modelom, ktorý je zobrazený na obrázku 5.2.

[5] Anglicky broadcast channels

[6] V zdrojovom kóde definovaný ako D[id] pre danú úlohu



Obrázok 5.2: Vzorový model plánovača

Vzorový model plánovača obsahuje 5 stavov, ktorými úloha môže prechádzať. V počiatočnom stave Free sa v danom momente môže nachádzať práve jedna úloha, preto je tento stav namodelovaný ako Commit. Nasledovne je kontrolovaná fronta s úlohami, nad ktorým je volaná funkcia *isEmpty(),* ktorá frontu kontroluje či je v danom momente prázdna.

```
bool isEmpty(){ return len == 0; }
```

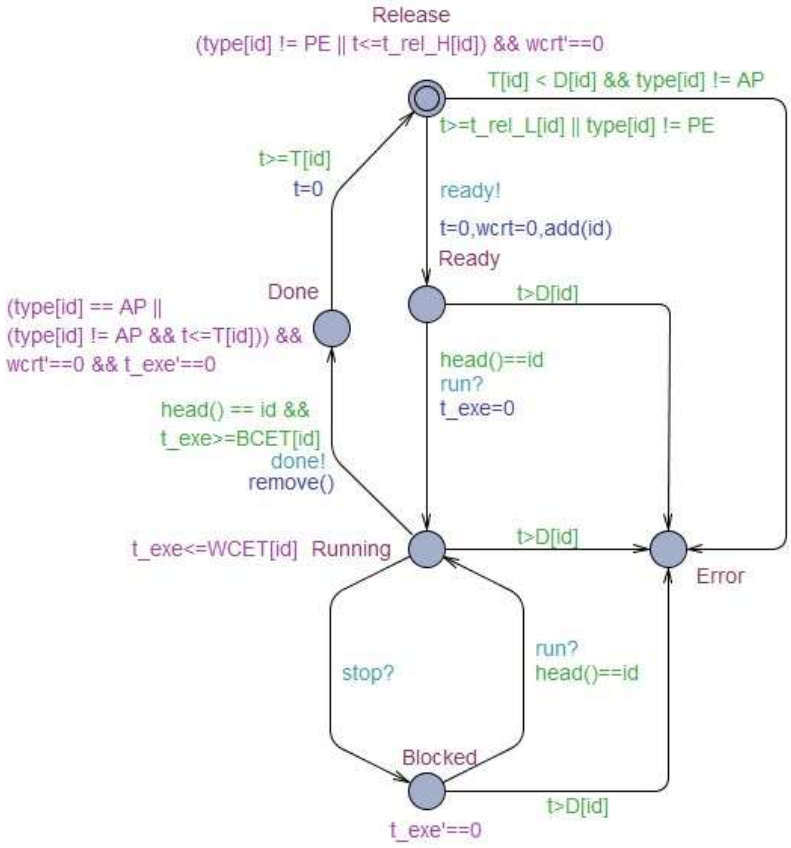
V prípade, že nie je zoznam úloh prázdny, plánovač prechádza priamo do stavu *Select.* V opačnom prípade je vyvolaný synchronizačným kanálom *ready?* signál, na ktorý potrebným spôsobom reaguje model úlohy. Stav *Select* je podobne ako *Free* modelovaný ako *commit*, takže úloha vybraná na vykonávanie môže byť v danom momente práve jedna. Pomocou synchronizačného kanála *run!* je modelom úlohy vyvolaný beh úlohy a presun v rámci modelu plánovača zo stavu *Select* do stavu *Occ*[7]. Ak sa v stave *Occ* dostane do modelu úloha s vyššou prioritou, ktorá odsunie aktuálne bežiacu úlohu, tak úloha, ktorá bola vykonávaná je presunutá pomocou synchronizačného kanála *stop!* do stavu *Blocked.* Po uvoľnení sa zablokovaná úloha dostáva späť ku vykonávaniu a svoju činnosť ukončuje prechodom z *Occ* do *Free* prostredníctvom komunikácie cez kanál *done?.*

5.2 Plánovanie podľa priorít úloh

V tomto prípade sa mechanizmus riadi podľa priorít, ktoré sú priradené každej úlohe a sú definované užívateľom v deklaračnej časti nástroja UPPAAL. Priority sú definované prostredníctvom celých čísel, kde platí pravidlo, že vyššie číslo má vyššiu prioritu.

Samotný model je skonštruovaný na základe vzorového modelu, ktorý sme si predstavili v podkapitole 5.1. Model je znázornený na obrázku 5.3, ktorý vidíme nižšie. Vzorový model bol doplnený o nejaké konštrukcie, ktoré si teraz vysvetlíme. V stave *Release* je definovaný invariant, ktorý nám určuje, že periodická úloha môže v tomto stave zostať po dobu, ktorá je určená jittrom pri zadefinovaní úlohy. Aperiodické a sporadické úlohy môžu v tomto stave zostať ľubovoľne dlhú dobu.

[7] Z anglického slova occurance, čiže výskyt



Obrázok 5.3: Mechanizmus podľa priraďovania priorít

Tomu sú prispôsobené aj jednotlivé stráže definované či už na prechode do stavu *Error* alebo *Ready*. Z *Release* do stavu *Error* sa úloha dostane v prípade, že periodickej úlohe je definovaná väčšia dĺžka deadlinu ako samotná perióda úlohy. Potom sa úloha posúva do stavu *Ready*. Tento prechod je zabezpečený definovaním synchronizačného kanála *Ready?,* ktorý zabezpečuje presun do stavu *Select* na strane plánovača. Pred tým ako úloha nadobudne stav *Ready,* je priradená do fronty úloh, ktoré budú vykonávané. Toto zabezpečuje funkcia *add(),* ktorej implementácia je zobrazená nižšie.

```
void add(pid_t id)
{
       pid_t i, tmp;
       queue[len] = id;
       for(i = len ; i > 0 && P[queue[i]] > P[queue[i-1]]; --i)
       {
         tmp = queue[i];
         queue[i] = queue[i-1];
         queue[i-1] = tmp;
       }
       len++;
}
```

Táto implementácia zabezpečuje jednak pridanie úlohy do vytvorenej fronty, ale taktiež kontroluje prioritu úlohy, ktorú chceme pridať, oproti ostatním úlohám, ktoré sa nachádzajú vo fronte. Ak má aktuálna úloha vyššiu prioritu ako úloha, ktorá bola do fronty pridaná pred ňou, tak si s ňou vymieňa miesto vo fronte.

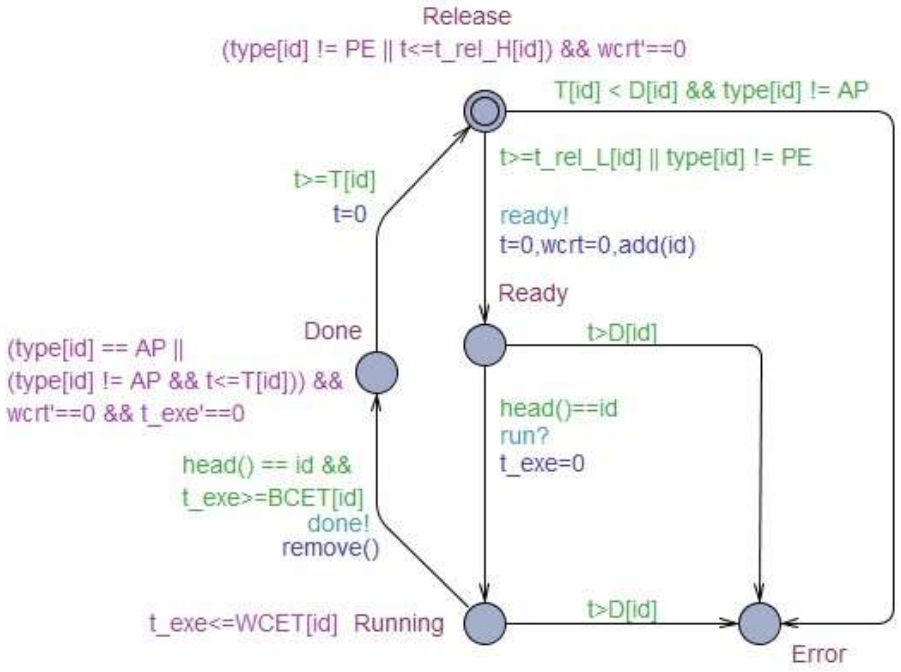
Prechod cez stav *Running* do *Done* je rovnaký ako vo vzorovom modeli. Je využívaný synchronizačný kanál *run?* na synchronizáciu s modelom plánovača. V prípade, že priorita aktuálne bežiacej úlohy je menšia ako priorita úlohy, ktorá prechádza zo stavu *Ready*, je aktuálne bežiaca úloha presunutá do stavu *Blocked.* Podobne ako vo vzorom modeli sú na presune do stavu *Done* využívané kanály *run?, done!* a taktiež funkcia *Remove()* s premennou *t_exe,* ktorá nás informuje o dĺžke času vykonávania tejto úlohy. Zmena oproti vzoru je v stave *Done.* V tomto stave je definovaný *invariant,* ktorý zabezpečuje, že aperiodická úloha môže v tomto stave zostať ľubovoľne dlhú dobu. Oproti tomu sporadické úlohy môžu tento stav opustiť v prípade, keď je splnená podmienka minimálnej periódy aktuálnej úlohy. Periodické úlohy musia počkať, pokiaľ sa nenaplní perióda. Prechody do stavu *Error* sa využívajú tak ako sme si ich vysvetlili v podkapitole 5.1.

V prípade modelu plánovača sa model zhoduje so vzorovým modelom, ktorý je zobrazený na obrázku 4.2 v podkapitole 5.1.

5.3 Mechanizmus plánovania FIFO

Tento mechanizmus sa neriadi princípom porovnávania priorít ako tomu bolo v predchádzajúcom prípade. Napriek tomu je základná konštrukcia veľmi podobná modelu, ktorý sa riadil plánovaním na základe priraďvania priorít jednotlivým úlohám v samotnom systéme.

Model úlohy plánovanej podľa mechanizmu FIFO si vysvetlíme na obrázku 5.4, ktorý môžeme vidieť nižšie.

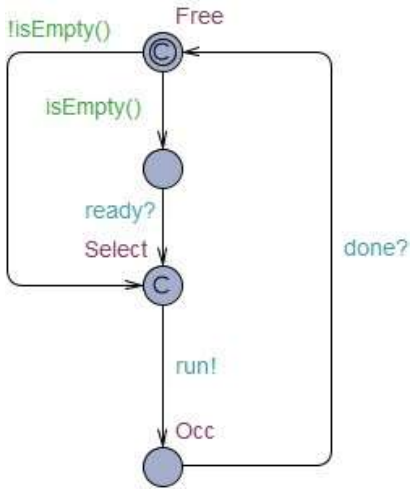


Obrázok 5.4: Model mechanizmu plánovania úlohy FIFO

Samotný model sa od predchádzajúceho prípadu veľmi nelíši. Podstatný rozdiel je vidieť na odstránení stavu *Blocked* z modelu, ktorý je zobrazený na obrázku 5.3. Tento stav je odstránený, pretože plánovač FIFO je nepreemptívny a bežiaca úloha nemôže byť prerušená. Naviac bola upravená funkcia *add()*, ktorá nepridáva úlohy do fronty na základe priority, ale podľa poradia požiadaviek na spustenie úlohy.

```
void add(pid_t id)
{
       pid_t i, tmp;
       queue[len] = id;
       len++;
}
```

Oproti vzorovému modelu plánovača sa pri mechanizme FIFO model líši tým, že je odstránený stav, do ktorého sa dostaneme zo stavu *Occ*. Tento stav je nadobudnutý, keď je aktuálne vykonávaná úloha presunutá do stavu *Blocked* v rámci modelu úlohy. Tento stav je využívaný v prípade, keď je nutné vymeniť aktuálne bežiacu úlohu, za úlohu, ktorá sa nachádzala na vrchole prioritnej fronty.



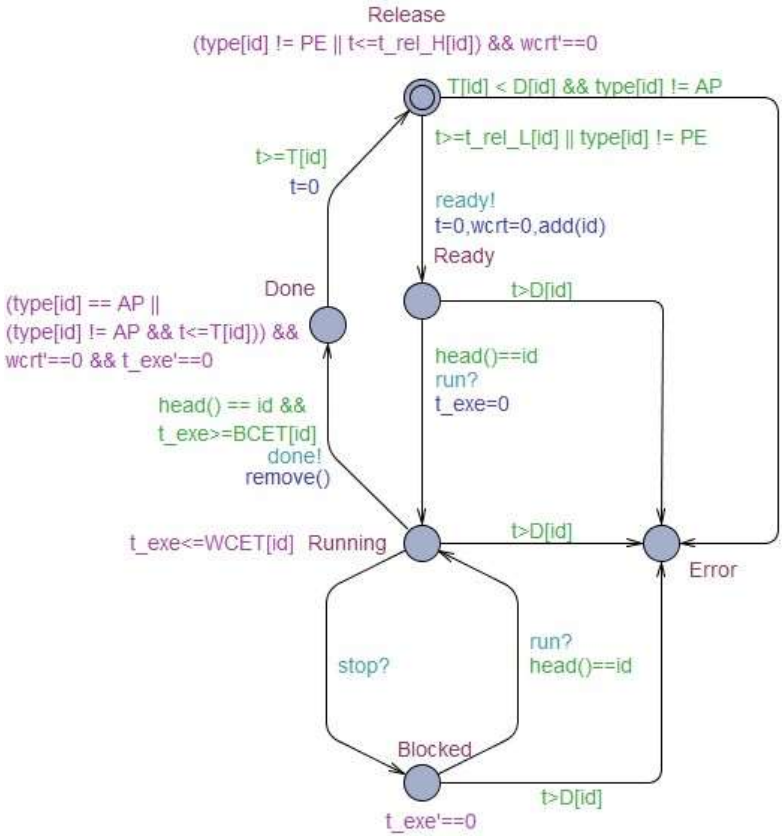
Obrázok 5.5: Model plánovača mechanizmu FIFO

5.4 Mechanizmus plánovania RR

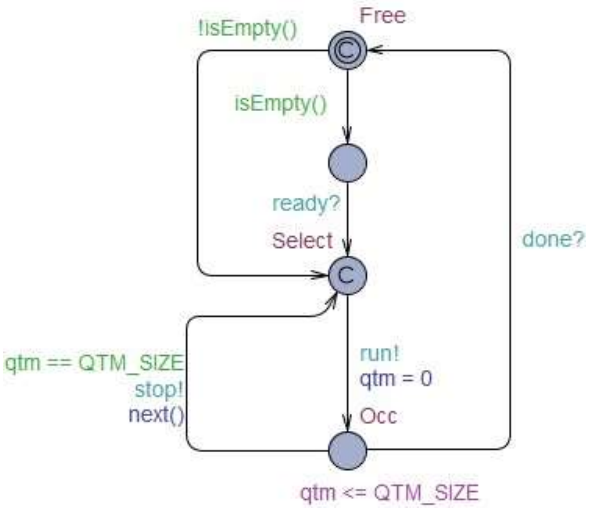
Mechanizmus Round Robin sme schopný využiť ako na periodické, tak aj na sporadické a aperiodické úlohy. Aj v tomto prípade budeme len minimálne zasahovať do základnej kostry, ktorú sme si vytvorili v podkapitole 5.1.

V konečnom dôsledku sa jedná o presnú kópiu modelu, ktorý sme vytvorili na plánovanie podľa priorít úloh. Aj tu je však model doplnený o nejaké tie zmeny. Základná podstata tohto mechanizmu spočíva v určení tzv. časového kvanta pre každú úlohu. Pridávanie úloh do fronty je taktiež rovnaké ako v mechanizme plánovania FIFO, pomocou funkcie *add()*. Na obrázku 5.6 si môžeme pozrieť model úlohy mechanizmu Round Robin.

```
void add(pid_t id)
{
      pid_t i, tmp;
      queue[len] = id;
      len++;
}
```



Obrázok 5.6: Mechanizmus plánovania úloh RR



Obrázok 5.7: Model plánovača mechanizmu RR

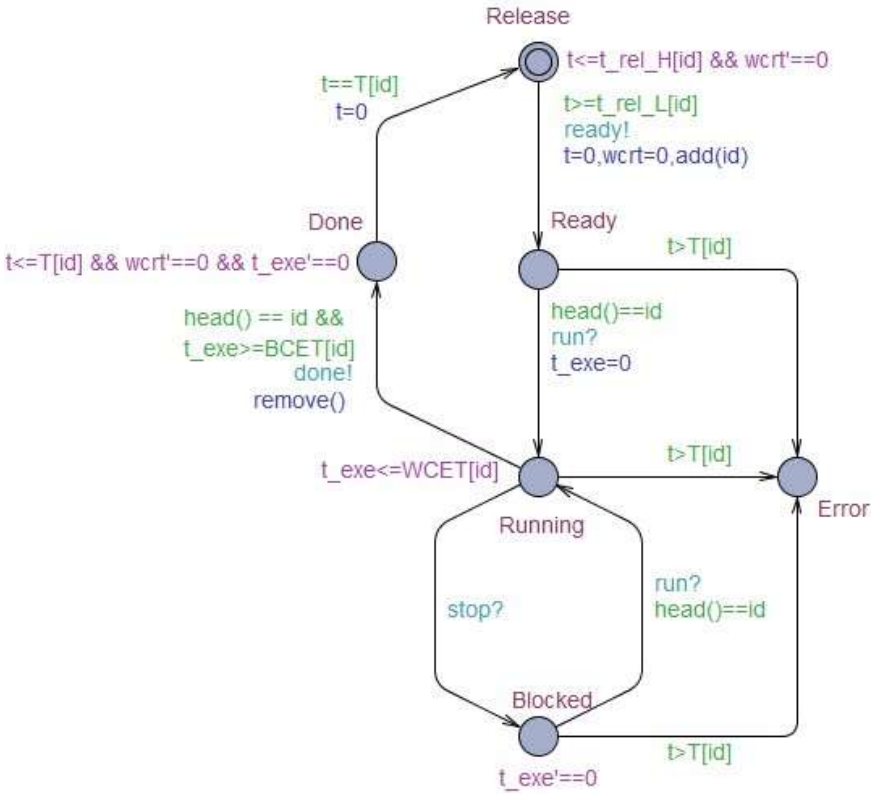
Model plánovača sa líši od modelu v predošlých kapitolách využívaním tzv. časového kvanta. Časové kvantum je v deklaračnej časti modelu definované ako *clock qtm*. Konštanta *QTM_SIZE* je deklarovaná užívateľom a je nastavená na hodnotu, ktorú si zvolí podľa vlastnej vôle. V prípade nedodržania tohto časového kvanta je pridaný prechod zo stavu *Occ* do stavu *Select*. V tomto prípade je volaná funkcia *next()*, ktorá je zobrazená nižšie.

Keď dôjde k vypršaniu časového kvanta, úloha je odstránená z fronty a je umiestnená späť do tejto fronty ako posledná. A práve túto funkcionalitu vykonáva vyššie zobrazená funkcia *next()*. Vo všeobecnosti táto funkcia zabezpečuje rotáciu úloh vo fronte.

```
void next()
{
      pid_t tmp = head();
      remove();
      add(tmp);
}
```

5.5 Mechanizmus plánovania RM

Keď porovnáme mechanizmus Rate Monotonic s predchádzajúcimi mechanizmami, tak ako hlavný rozdiel by sme mali spomenúť, že tento mechanizmus nepodporuje plánovanie aperiodických a sporadických úloh, pretože vyžaduje aby boli úlohy periodické.



Obrázok 5.8: Model úlohy mechanizmu RM

Podstatný rozdiel je ten, že v prípade RM úlohy je deadline nastavený na dobu periódy, čo je dôvod prečo model úlohy neobsahuje prechod zo stavu *Release* do stavu *Error*. Z toho vyplýva, že je tak isto upravený invariant v stave *Realease,* kde sme z podmienky odstránili kontrolu či je úloha periodická alebo nie. Zmena bola tak isto urobená v stave *Done*, kde sme z podmienky odstránili kontrolu či je úloha aperiodická. Podobne je zmenená stráž na prechode do stavu *Ready,* nakoľko mechanizmus nepočíta so sporadickými a aperiodickými úlohami. Taktiež bola upravená funkcia *add()* podľa hlavného princípu mechanizmu Rate Monotonic.

Funkcia *add()* bola upravená tak, aby úlohy s nižšou periódou boli uprednostnené pred úlohami s vyššou periódou. Tým je dosiahnutá implementácia odpovedajúca mechanizmu plánovania Rate Monotonic.

```
void add(pid_t id)
{
      pid_t i, tmp;
      queue[len] = id;
      for(i = len ; i > 0 && T[queue[i]] < T[queue[i-1]]; --i)
      {
           tmp = queue[i];
           queue[i] = queue[i-1];
           queue[i-1] = tmp;
      }
      len++;
}
```

Model plánovača v prípade mechanizmu Rate Monotonic je taký istý, ako bol vo vzorom prípade v podkapitole 5.1.

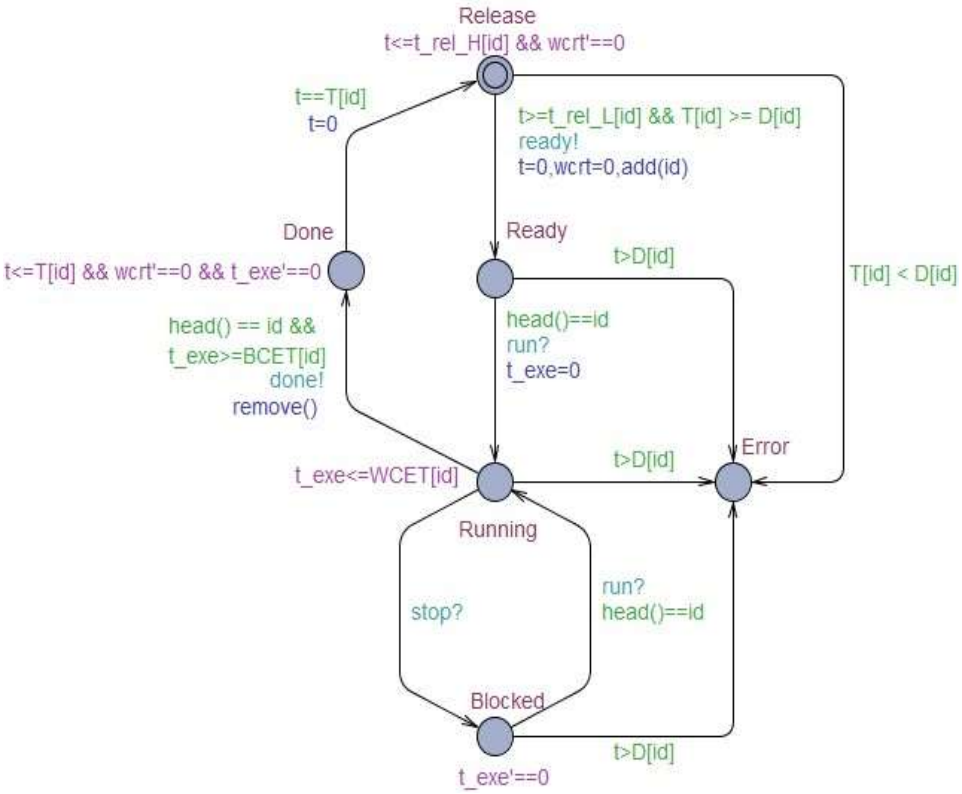
5.6 Mechanizmus plánovania DM

Model mechanizmu úlohy Deadline Monotonic je takmer zhodný s modelom, ktorý je zobrazený na obrázku 5.8. Pre porovnanie je zobrazený na obrázku 5.9. Vysvetlíme si, v čom spočíva hlavný rozdiel medzi týmito dvoma mechanizmami, aj keď oba modely vyzerajú na prvý pohľad veľmi podobne.

Model tak isto ako pri mechanizme RM neimplementuje aperiodické a sporadické úlohy, pretože mechanizmus počíta iba s plánovaním periodických úloh. Oproti predchádzajúcemu modelu mechanizmu bol doplnený prechod zo stavu *Release* do stavu *Error* z dôvodu, ktorý sme si vysvetlili v kapitole 5.1 pri vzorovom riešení.

Funkcia *add()* je taktiež upravená ako tomu bolo v predchádzajúcich prípadoch. Pri tomto modeli, funkcia pridáva úlohy na začiatok fronty na základe pravidla najmenšieho deadlinu. Tým je dosiahnutá implementácia, ktorá odpovedá chovaniu plánovacieho algoritmu Deadline Monotonic.

```
void add(pid_t id)
{
  pid_t i, tmp;
  queue[len] = id;
  for(i = len ; i > 0 && D[queue[i]] < D[queue[i-1]]; --i)
  {
    tmp = queue[i];
    queue[i] = queue[i-1];
    queue[i-1] = tmp;
  }
  len++;
}
```



Obrázok 5.9: Model úlohy mechanizmu DM

Model plánovača mechanizmu DM je taký istý ako sme si ho opisovali podľa obrázku v podkapitole 5.1 na obrázku 5.2.

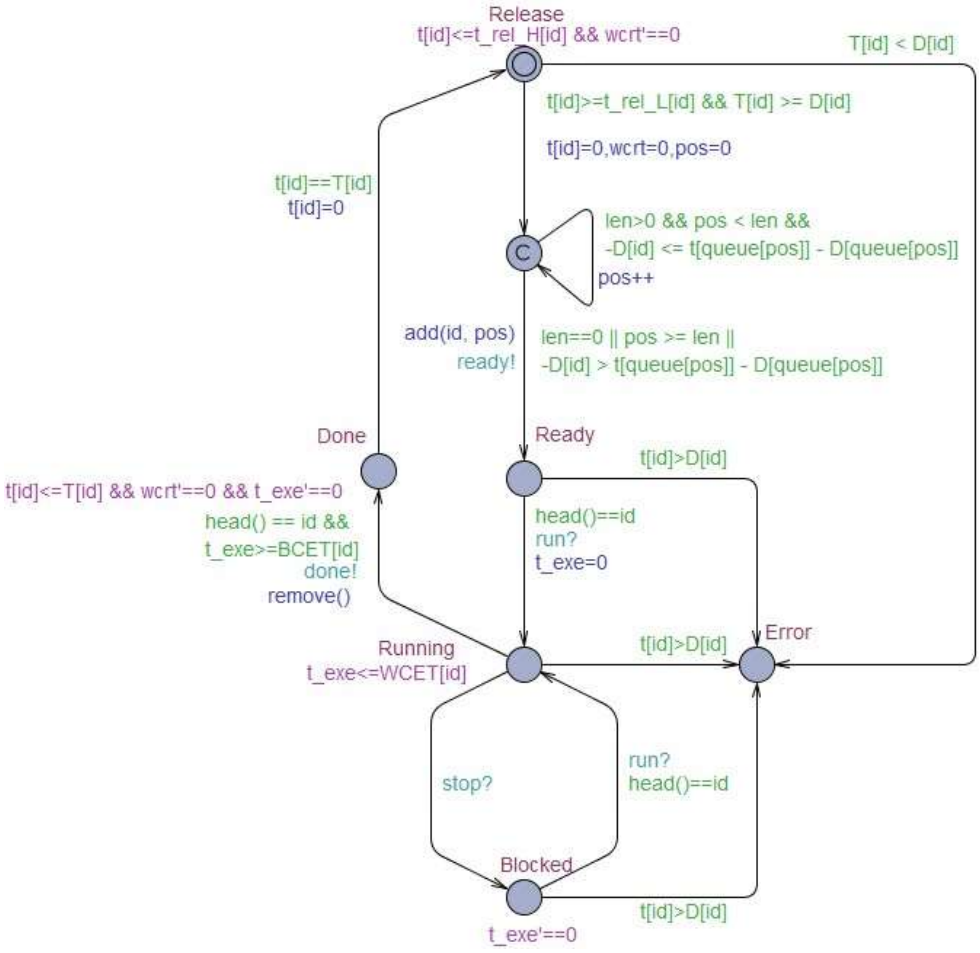
5.7 Mechanizmus plánovania EDF

Mechanizmus rieši tak ako v predchádzajúcich dvoch prípadoch iba plánovanie periodických úloh, a so sporadickými a aperiodickými úlohami nepočíta. Ale bolo nutné urobiť väčšiu zmenu, čo sa týka modelu, nakoľko pridávanie úloh do fronty je komplikovanejšie ako v predchádzajúcich mechanizmoch.

Rozšírený model mechanizmu Earliest Deadline First je zobrazený na obrázku 5.10. Pre modelovanie tohto plánovacieho algoritmu je potrebné pridať jeden stav medzi stavy *Ready* a *Release.* Cyklus, ktorý je implementovaný v danom stave testuje, či ostávajúci čas do deadlinu úlohy na pozícii *pos* vo fronte, je vyšší než deadline novo spustenej úlohy. S každou iteráciou sa zvyšuje hodnota premennej *pos*. Premennú *pos* sme si zadefinovali ako pomocnú premennú.

Vo chvíli, keď má úloha vo fronte na pozícii *pos* deadline vzdialenejší než nová úloha, tak potom premenná *pos* udáva miesto vo fronte, kam má byť nová úloha zaradená. Pre správnu funkčnosť musela byť upravená funkcia *add(),* ktorá vkladá novú úlohu do fronty na pozíciu zadanú parametrom funkcie.

```
void add(pid_t id, pid_t pos)
{
	pid_t i, tmp;
	queue[len] = id;
	for(i = len ; i > pos; --i)
	{
		tmp = queue[i];
		queue[i] = queue[i-1];
		queue[i-1] = tmp;
	}
	len++;
}
```



Obrázok 5.10: Model úlohy mechanizmu EDF

Model plánovača pri modelovaní algoritmu EDF je taký istý ako v predchádzajúcom prípade.

6 Overenie vlastností modelov

V tejto kapitole by sme sa zamerali na konečné vyhodnotenie a porovnanie výsledkov, ktorých sme docielili pomocou nástroja UPPAAL. Nástroje, ktoré budeme používať na porovnanie, sa nazývajú TimesTool[8] a Cheddar. Používajú sa na modelovanie a plánovanie na základe typu plánovača úloh, s ktorými je simulácia spustená. Sme schopný modelovať či už periodické, tak aj sporadické a aperiodické úlohy. Ale my sa budeme zaoberať vyložene plánovaním periodických úloh, nakoľko porovnávanie plánovania sporadických a aperiodických úloh by overovanie výsledkov výrazne skomplikovalo. V nasledujúcich podkapitolách budeme výsledky porovnávať na základe grafických výstupov nástrojov. V nástroji UPPAAL budeme výstup zobrazovať pomocou Ganttovho diagramu. Čitateľovi je vhodné vysvetliť, ako sa jednotlivé stavy zobrazujú v grafoch. Každý stav je v grafe možné rozoznať na základe farby (zelená – *Ready,* modrá – *Running,* fialová – *Blocked*, oranžová – *Error*). Jednotlivé podkapitoly si rozdelíme podľa mechanizmov, ktoré sme využili. Verzie programov TimesTool a Cheddar, ktoré boli použité na generovanie grafov a overovanie výsledkov sú sprístupnené na školskom servery.

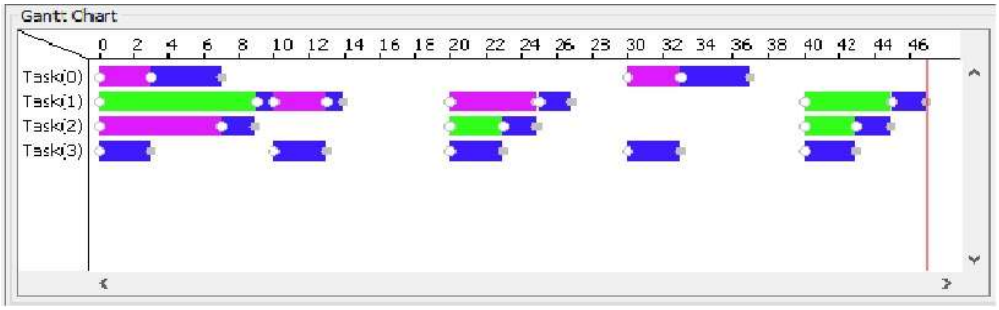
6.1 Plánovanie podľa priorít úloh

Upravený model, ktorý bol prispôsobený okrem plánovania periodických úloh aj na plánovanie sporadických a aperiodických úloh bol testovaný na vzorke štyroch úloh so zadanými parametrami. Cieľom zadania parametrov bolo získať výstupy, na základe ktorých by sme vedeli poukázať na priebeh plánovania a vykonávania jednotlivých úloh na dvoch rozličných nástrojoch. Parametre, ktoré boli zvolené na testovanie sú zobrazené v tabuľke 6.1.

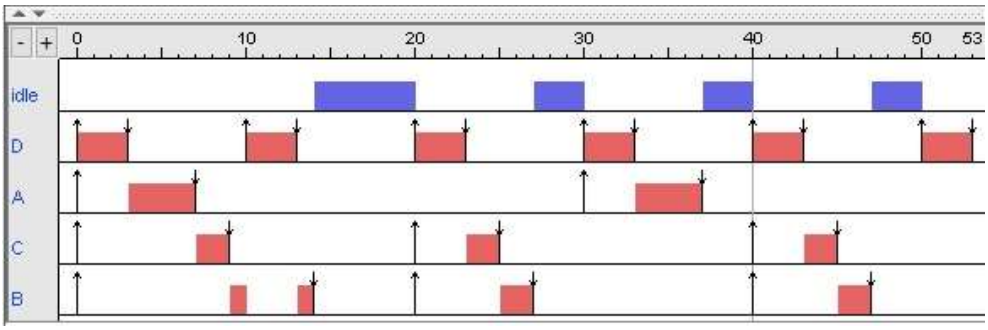
Úloha	P	C	D	T
A	3	4	25	30
B	1	2	20	20
C	2	2	15	20
D	4	3	8	10

Tabuľka 6.1: Parametre úloh mechanizmu priraďovania priorít

[8] Z angličtiny A Tool for Modeling and Implementation of Embedded Systems



Obrázok 6.1: Graf plánovania podľa priorít úloh v UPAAAL



Obrázok 6.2: Graf plánovania podľa priorít v TimesTool

Grafy plánovania a priebehu jednotlivých úloh sú zobrazené na obrázkoch 6.1 a 6.2[9]. Na grafoch môžeme vidieť, že priebehy úloh sa v obidvoch prípadoch zhodujú. Na Ganttovom diagrame sú veľmi prehľadne zobrazené úseky, keď sú úlohy v stave *Blocked*. Tento stav nastáva práve vtedy, keď nová úloha, ktorá ma vyššiu prioritu je uprednostnená pred úlohou, ktorá bola vykonávaná. Túto situáciu si môžeme všimnúť v čase 10, keď úloha *Task(3)*, ktorá v tomto čase príde a má vyššiu prioritu, preruší vykonávanie úlohy *Task(1).* Úloha *Task(1)* sa v tomto momente presúva do stavu *Blocked,* a v čase 13, keď úloha *Task(3)* ukončí svoje vykonávanie sa môže vrátiť späť do stavu v akom skončila. V rozmedzí od času 14 do 20 sa vo fronte nenachádzajú žiadne úlohy a čaká sa na periódu, ktorá je v prípade troch úloh nastavená na čas 20. Doba čakania jednotlivých úloh vo fronte závisí od dĺžky vykonávania úlohy s najvyššou prioritou. Problém by mohol napríklad nastať, keď by mala úloha s najvyššou prioritou vo svojom tele nekonečný cyklus. V tejto situácii by mohlo dôjsť ku komplikáciám, nakoľko túto úlohu by iná úloha nemohla prerušiť. Ako zaujímavý štatistický údaj môžeme uviesť najhoršie doby zotrvania úloh v systéme, ktoré boli vygenerované nástrojom TimesTool.

Úloha	A	B	C	D
WCRT	7	14	9	3

Tabuľka 6.2: Hodnoty WCRT jednotlivých úloh

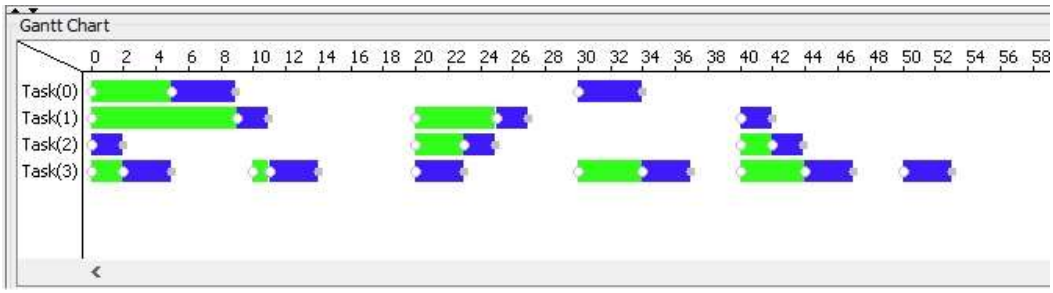
[9] Úlohy 0,1,2,3 v UPPAAL a úlohy A,B,C,D v TimesTool sa v tomto poradí zhodujú a sú totožné len poradie ich zobrazenia v grafoch je rôzne.

6.2 FIFO plánovanie

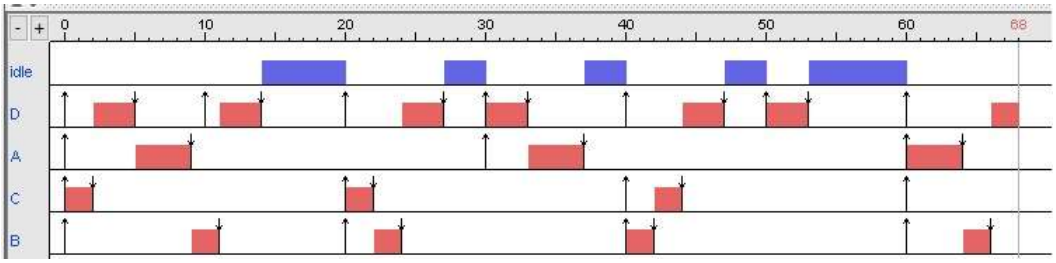
V prípade tohto mechanizmu využijeme sadu úloh, ktorú sme využili pri zvolení parametrov v predošlom mechanizme. Pri zvolení parametrov tentokrát vynecháme hodnotu určenia priority nakoľko poradie vykonávania úloh je dané príchodom úloh do fronty. Plánovanie podľa mechanizmu FIFO zabezpečuje, že každá úloha, ktorá príde do systému dostane čas, ktorý je jej pridelený na vykonávanie a nemôže nastať situácia, keď je úloha vynechávaná kvôli tomu, že ju vždy preskočí iná s vyššou prioritou.

Úloha	C	D	T
A	4	25	30
B	2	20	20
C	2	15	20
D	3	15	15

Tabuľka 6.3: Parametre úloh mechanizmu FIFO



Obrázok 6.3: Graf plánovania mechanizmu FIFO v UPPAAL



Obrázok 6.4: Graf plánovania mechanizmu FIFO v TimesTool

Tak ako sme už vyššie naznačili, pri plánovaní typu FIFO sa jedná o nepreemptívne plánovanie, nakoľko úloha nemôže byť počas behu prerušená. Z toho dôvodu bol odstránený stav *Blocked* z modelu úlohy, a stav z modelu plánovača, ktorý zabezpečoval výmenu aktuálne bežiacej úlohy, za úlohu, ktorá sa nachádzala na vrchole fronty. Preto na grafe nedochádza k prepínaniu kontextov medzi úlohami ako na obrázku 6.1. Na obrázku 6.3 je zobrazený graf, kde môžeme vidieť, že úlohy sú vykonávané podľa poradia v akom prišli do fronty. Môžeme pozorovať, že úlohy dodržiavajú periódy, ktoré im boli pridelené v podobe parametru T tak ako v predošlom prípade. Medzi problémy tohto mechanizmu môžeme zaradiť príklad keď sa úloha s dlhšou dobou mechanizmu dostane pred úlohu s kratšou dobou mechanizmu. To ešte nemusí spôsobiť až také zlé následky na konečný beh systému, no môže to mať vplyv na efektivitu vykonávania úloh. Ďalším problémom môže byť prípad, keď aktuálne bežiaca úloha obsahuje nekonečný cyklus, nakoľko tento

mechanizmus je nepreemptívny. Výsledkom tohto problému môžu byť následky, ktoré môžu spôsobiť až zastavenie behu systému. Pri parametroch, s ktorými sme úlohy spustili môže dôjsť k situácii, že úlohe *D* vyprší deadline. Táto situácia nastane v čase 68 na obrázku 6.4. Ak zmeníme parametre úloh, a nastavíme periódu úlohy *D* na hodnotu 15 a hodnotu deadlinu tiež na 15, úlohy by mali byť plánovateľné. Po zmene nám nástroj TimesTool vygeneroval hodnoty WCRT uvedené nižšie v tabuľke 6.4.

Úloha	A	B	C	D
WCRT	11	11	11	11

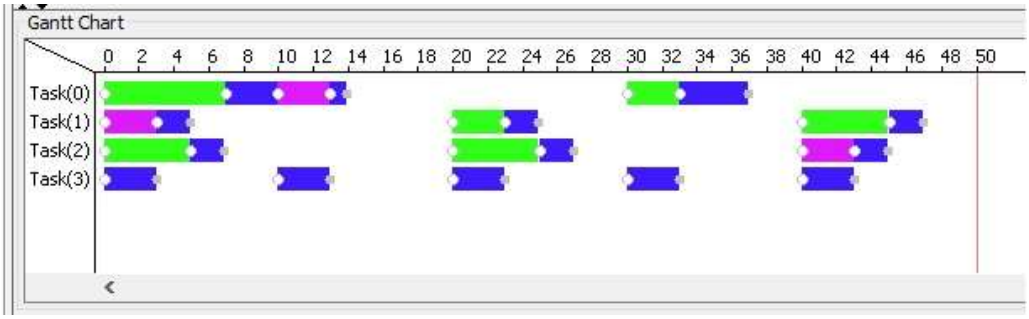
Tabuľka 6.4: Parametre úloh mechanizmu FIFO

6.3 RM plánovanie

Prideľovanie priorít pri tomto mechanizme funguje na princípe, že čím je častejšie úloha volaná, tým je jej pridelená vyššia priorita. Lenže aj tu môže dochádzať k zásadným problémom, ako napríklad, keď je procesor pridelený úlohe až po vypršaní jej rozsahu. Preto sa tento mechanizmus nemusí hodiť v niektorých prípadoch plánovania úloh. Parametre úloh sme použili ako v minulých prípadoch, akurát bol vynechaný parameter, ktorý určuje deadline úlohy, nakoľko sa v tomto prípade rovná perióde úlohy.

Úloha	C	T
A	4	30
B	2	20
C	2	20
D	3	10

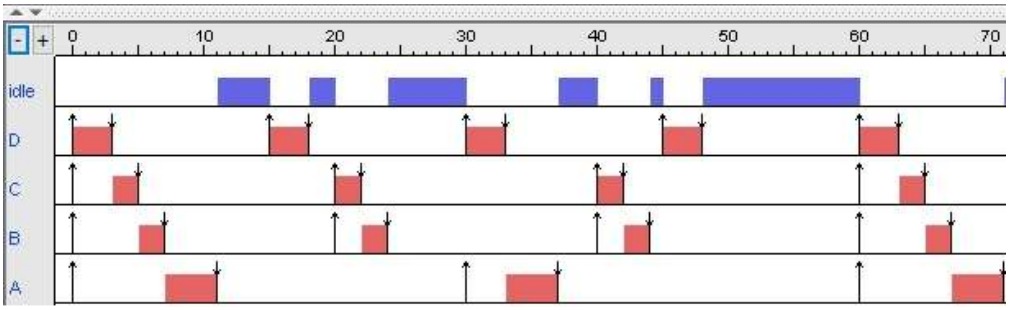
Tabuľka 6.5: Parametre úloh mechanizmu RM



Obrázok 6.5: Graf plánovania mechanizmu RM V UPPAAL

Riešením tejto situácie môže byť zmena parametrov úlohy, tak aby bola úloha plánovateľná. Avšak táto metóda nie je doporučovaná, nakoľko dodatočná zmena parametrov úlohy môže spôsobiť zmenu špecifikácie celého systému. Druhým riešením tohto problému je zmena plánovacieho mechanizmu, ktorý by bol vhodnejší na plánovanie daných úloh. Na grafoch úloh, ktoré sú zobrazené na obrázkoch 6.5 a 6.6 môžeme vidieť, že úloha, ktorá je najčastejšie vykonávaná je úloha *Task(3),* pretože je najčastejšie volaná. Naopak úloha *Task(0)* je do systému volaná najmenej často. V čase 10 je tejto úlohe procesor odobraný, pretože do systému prišla úloha *Task(3)*, ktorá ma pred úlohou

Task(0) prioritu. Potom čo *Task(3)* dokončí svoje vykonávanie je procesor vrátený úlohe *Task(0)*. Podľa tabuľky hodnôt WCRT 6.6, môžeme povedať, že najhoršia doba úloh v systéme sa v porovnaní s mechanizmom plánovania na základe priorít a mechanizmom FIFO zmenila. Na základe týchto hodnôt by sa dalo usúdiť, že plánovanie podľa tohto mechanizmu sa zdá byť efektívnejšie z pohľadu úloh, ktoré systém zaťažujú najčastejšie a naopak.



Obrázok 6.6: Graf plánovania mechanizmu RM v TimesTool

Úloha	A	B	C	D
WCRT	14	7	5	3

Tabuľka 6.6: Hodnoty WCRT jednotlivých úloh

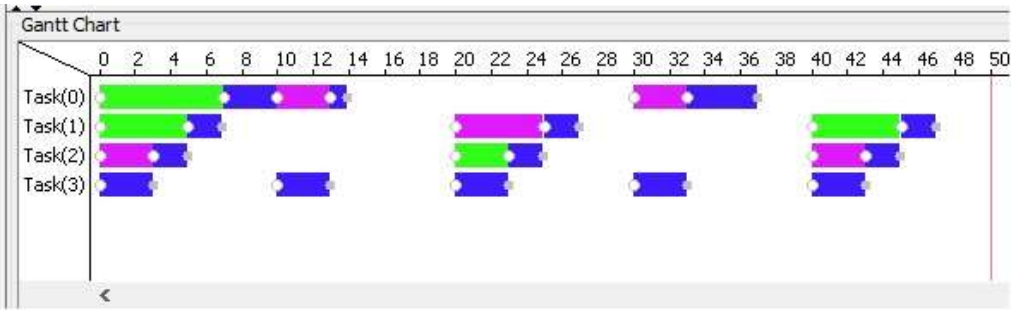
6.4 DM plánovanie

Mechanizmus Deadline Monotonic je použiteľný v mnohých prípadoch, keď sa Rate Monotonic využiť nedá. Základom implementácie DM mechanizmu je, že najvyššiu prioritu má tá úloha, ktorej deadline je najbližšie. Okrem toho sa modely mechanizmov RM a DM veľmi podobajú aj keď podstata princípu plánovania je iná. Na testovanie boli použité hodnoty parametrov, ktoré sme využívali už v predošlých mechanizmoch.

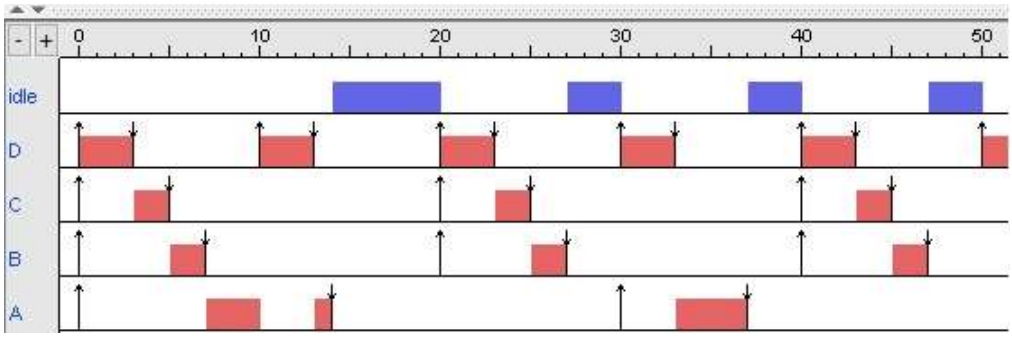
Úloha	C	D	T
A	4	25	30
B	2	20	20
C	2	15	20
D	3	8	10

Tabuľka 6.7: Parametre úloh mechanizmu DM

Po testovaní na rôznej sade úloh môžeme na našom modeli vytvorenom v nástroji UPPAAL potvrdiť, že v porovnaní s mechanizmom RM je mechanizmus DM optimálny na množine, kde je relatívne časové rozmedzie úlohy menšie ako samotná perióda. Priebeh grafov RM a DM je pri testovaní s danou sadou parametrov veľmi podobný. Jediný rozdiel je ten, že úloha *Task(2)* sa spustí skôr ako *Task(1),* pretože jej deadline je naplánovaný skôr. Pokiaľ by sme mechanizmy porovnávali na základe hodnôt WCRT tak podľa tabuľky 6.8 by sme zistili, že hodnoty, ktoré boli získané metódou DM nám dávajú veľmi podobné hodnoty.



Obrázok 6.7: Graf plánovania mechanizmu DM v UPPAAL



Obrázok 6.8: Graf plánovania mechanizmu DM v TimesTool

Úloha	A	B	C	D
WCRT	14	7	5	3

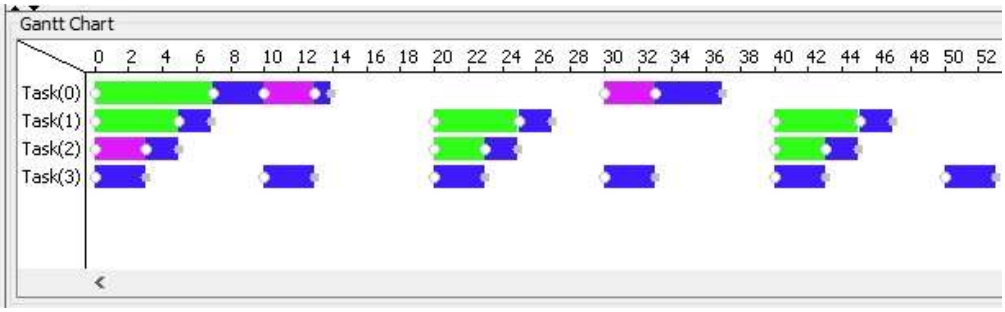
Tabuľka 6.8: Hodnoty WCRT jednotlivých úloh

6.5 EDF plánovanie

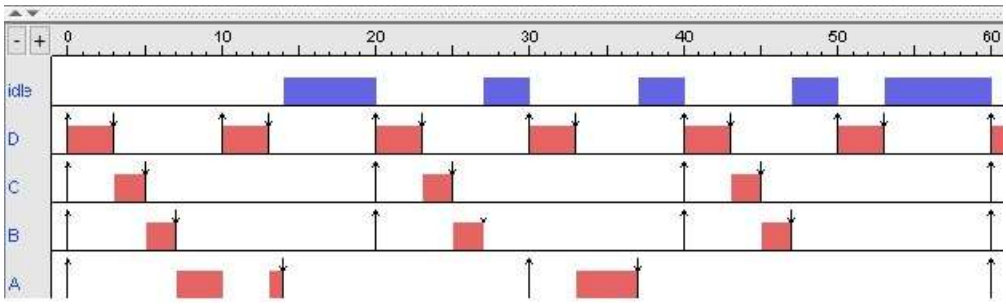
Plánovanie týmto spôsobom je založené na dynamickom priraďovaní priorít. Model mechanizmu Earliest Deadline First je od ostatných odlišný najmä tým, že je do modelu pridaný stav, ktorý určuje pozíciu úlohy vo fronte. Táto pozícia je určená hodnotou vzdialenosti k deadlinu danej úlohy. Priority sú vždy aktualizované v čase volania novej úlohy. Čím je táto hodnota menšia, tým je samotná priorita úlohy väčšia. Hodnoty parametrov, ktoré sme použili sú zobrazené na v tabuľke 6.9.

Úloha	C	D	T
A	4	25	30
B	2	20	20
C	2	15	20
D	3	8	10

Tabuľka 6.9: Parametre úloh mechanizmu EDF



Obrázok 6.9: Graf plánovania mechanizmu EDF v UPPAAL



Obrázok 6.10: Graf plánovania mechanizmu EDF v TimesTool

Na grafoch 6.9 a 6.10 môžeme vidieť, že plánovanie úloh je v oboch nástrojoch totožné. *Task(3)* je v systéme najdlhšiu dobu nakoľko deadline tejto úlohy je vždy, keď táto úloha nadobudne hodnotu novej periódy najbližšie. Naopak pri úlohe *A* respektíve *Task(1),* dochádza k situácii keď je táto úloha v systéme najmenej krát v porovnaní s ostatnými úlohami. Toto je zapríčinené tým, že perióda úlohy je rovná hodnote 30 a tak isto aj deadline úlohy (hodnota 25), je v porovnaní s ostatnými úlohami pomerne veľký, a preto je jej procesor pridelený v konečnom dôsledku najmenej krát. Pre porovnanie s mechanizmom DM, môžeme vidieť, že hodnoty WCRT sú na základe tabuľky 6.10 a 6.9 pri tejto sade testovacích úloh také isté. Na základe testovania môžeme povedať, že mechanizmus EDF je optimálny na množine nezávislých preemptívnych úloh, práve vtedy keď na tejto množine existuje prípustný plán. V tom prípade je mechanizmus EDF schopný tento plán nájsť.

Úloha	A	B	C	D
WCRT	14	7	5	3

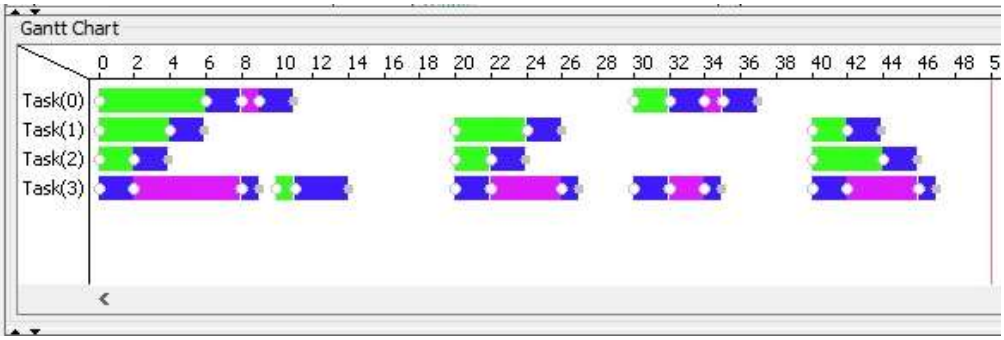
Tabuľka 6.10: Hodnoty WCRT jednotlivých úloh

6.6 RR plánovanie

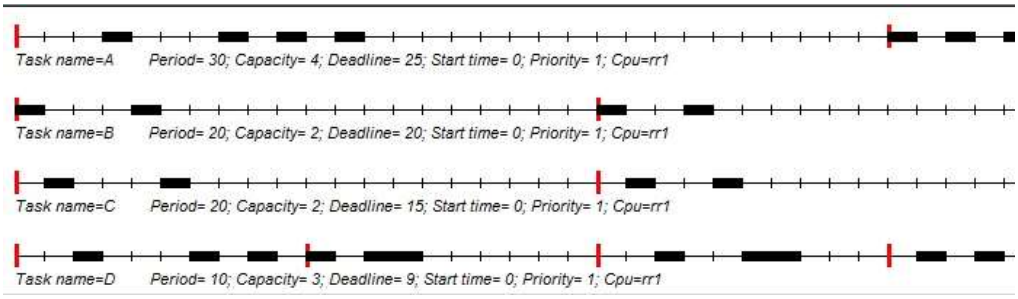
Princíp tohto mechanizmu je založený na určení veľkosti časového kvanta. Od tohto čísla sa odvíja, na aký dlhý čas môže byť procesor pridelený jednotlivým úlohám. Na testovanie plánovania tohto mechanizmu sme využili sadu úloh, ktorá je zobrazená v tabuľke 6.11, ktorá je zobrazená nižšie. Museli sme ju trochu pozmeniť nakoľko s parametrami, ktoré sme využívali v predošlých prípadoch, dochádza k situácii, keď úloha D nadobudne svoj deadline počas plynutia prvej periódy.

Úloha	C	D	T
A	4	25	30
B	2	20	20
C	2	15	20
D	3	9	10

Tabuľka 6.11: Parametre úloh mechanizmu RR



Obrázok 6.11: Graf plánovania mechanizmu RR v UPPAAL



Obrázok 6.12: Graf plánovania mechanizmu RR v Cheddare

Tak ako sme už spomínali na simulovanie mechanizmu Round Robin sme využili nástroj Cheddar, pretože plánovač tohto mechanizmu je už naimplementovaný v tomto nástroji. Ako môžeme sledovať na grafe 6.11 úlohy sú vykonávané v poradí v akom sú usporiadané do fronty. Každej úlohe je procesor pridelený na čas, ktorý je určený hodnotou časového kvanta. Môžu nastať dve situácie. Úloha môže skončiť skôr než uplynie časové kvantum a sama uvoľní CPU, ktoré je potom pridelené úlohe zo začiatku fronty. Druhá možnosť je tá, že úloha behom časového kvanta neskončí. Po uplynutí časového kvanta a odobraní procesora je jej beh prerušený. Úloha, ktorá bola prerušená je zaradená na koniec fronty a procesor je pridelený úlohe, ktorá sa nachádza na začiatku fronty. Tento prípad je znázornený na obrázku 6.11, kde je úloha *Task(3)* prerušená v čase 2 a je zaradená na koniec fronty. V momente, keď na ňu znova príde rad, je jej vykonávanie dokončené. Problém sa môže vyskytnúť aj pri stanovení veľkosti kvanta. Pri testovaní bolo zistené, že malé kvantum vedie na kratšiu dobu odozvy, ale nakoľko doba prepnutia kontextu je konštantná, réžia spojená s prepnutím kontextu narastá na úkor vlastnej činnosti úloh. Naopak veľké kvantum degraduje mechanizmus Round Robin na FIFO.

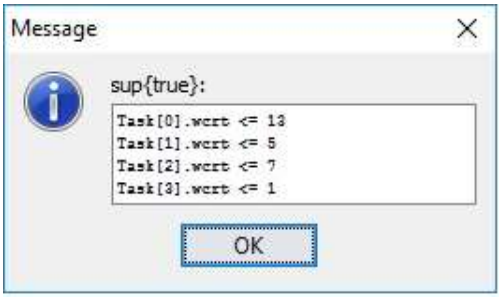
6.7 Testovanie vlastností modelu

V tejto podkapitole sa zameriame na otestovanie a dokázanie niektorých vlastností modelov. Zamerali by sme sa výhradne na jeden mechanizmus, na ktorom by sme ilustrovali testy, ktoré porovnáme s výstupmi z nástroja TimesTool. Skúsime si porovnať hodnoty WCRT pri zmene parametrov jednotlivých úloh. Hodnoty parametrov si zobrazíme v tabuľkách. Hodnoty WCRT v nástroji UPPAAL zisťujeme podľa verefikačného dotazu, ktorý je zobrazený nižšie.

```
sup: Task(0).wcrt , Task(1).wcrt, Task(2).wcrt, Task(3).wcrt
```

Úloha	C	T
A	5	50
B	4	20
C	2	30
D	1	10

Tabuľka 6.12: Parametre úloh mechanizmu RM

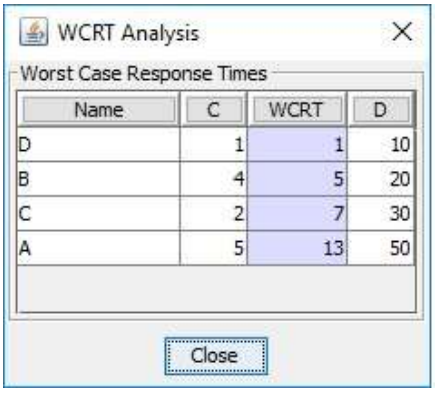


Obrázok 6.13: Hodnoty WCRT jednotlivých úloh

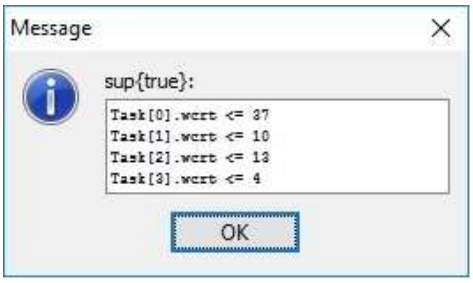
Úloha	C	T
A	7	40
B	6	20
C	3	30
D	4	15

Tabuľka 6.13: Parametre úloh mechanizmu RM

Po zmene parametrov zobrazených v tabuľkách 6.12 a 6.13 zistíme že hodnoty WCRT sa zmenili a sú závislé na zmene hodnôt parametrov *C* a *T*. V prípade mechanizmu Rate monotonic sa hodnota deadlinu rovná hodnote periódy úlohy. Na overenie správnosti vykonávania jednotlivých úloh sme využili nástroj TimesTool. Na obrázku 6.14 môžeme vidieť hodnoty WCRT vygenerované týmto nástrojom. Preto môžeme povedať že nami implementovaný model zodpovedá hodnotám nameraným v inom nástroji. Zmeny parametrov podľa tabuliek 6.12 a 6.13 spôsobia zmeny hodnôt WCRT jednotlivých úloh, ktoré môžeme sledovať na obrázku 6.15.



Obrázok 6.14: Hodnoty WCRT jednotlivých úloh z nástroja TimesTool



Obrázok 6.15: Hodnoty WCRT jednotlivých úloh

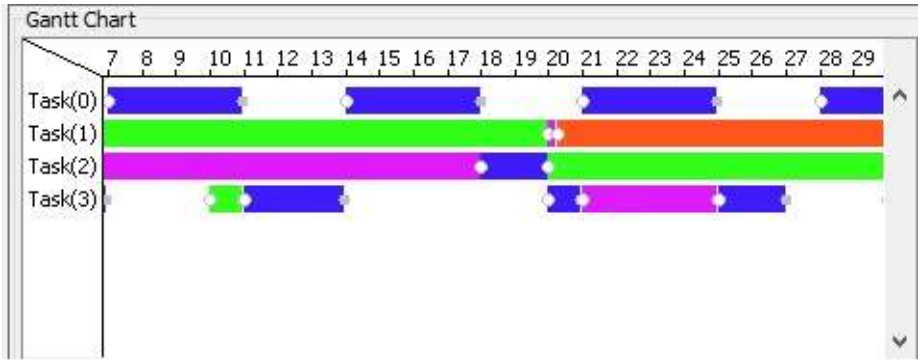
Ďalším dôležitým dotazom, ktorý by mal byť testovaný v každom modeli, je dotaz či môže nastať deadlock. Pri overení či sa v modeli mechanizmu RM, ktorý sme implementovali nachádza deadlock využijeme dotaz uvedený nižšie. V našom prípade sa dozvieme, že test na deadlock je negatívny.

```
A[] not deadlock
```

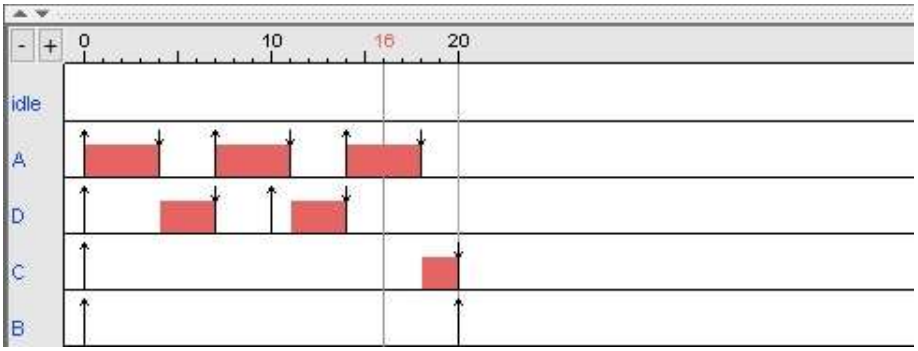
Ďalej si skúsime zobraziť situáciu, keď sa úloha dostane do stavu *Error* a ako sa to na našom výslednom diagrame prejaví. Tento prípad si ukážeme na obrázku 6.16, kde je zobrazený diagram z nástroja UPPAAL pri simulácii modelu s parametrami z tabuľky 6.14.

Úloha	C	T
A	4	7
B	2	20
C	2	20
D	3	10

Tabuľka 6.14: Parametre úloh mechanizmu RM



Obrázok 6.16: Graf plánovania mechanizmu RM V UPPAAL



Obrázok 6.17: Graf plánovania mechanizmu RM v TimesTool

Na grafe 6.16 môžeme vidieť, že úloha *Task(1)* sa dostáva do stavu *Error* v čase 20. V tomto čase je priorita priradená úlohe *Task(3)*, pretože jej prerióda je menšia, takže má vyššiu prioritu. Túto situáciu môžeme porovnať s obrázkom 6.17, kde sú úlohy spustené v nástroji TimesTool s tými istými parametrami ako v nástroji UPPAAL. Na obrázku 6.17 môžeme vidieť, že simulácia sa zastaví, pretože nie je plánovateľná z dôvodu, že úloha *B* sa dostáva v čase 20 do stavu *Error* a tým sa simulácia končí. Nadobudnutie stavu *Error* úlohou *Task(1)* môžeme overiť verifikačným dotazom zobrazeným nižšie. V prípade, že tento dotaz otestujeme na prípade zobrazenom na obrázku 6.16 dostaneme výstup „*Property may be satisfied*“. Na základe tohto výstupu môžeme povedať, že tento stav môže nastať.

```
E<> Task(2).Error
```

Testy sme v tejto podkapitole ilustrovali na mechanizme Rate Monotonic. Na základe výsledkov týchto testov a ich porovnaní s výsledkami z nástroja TimesTool môžeme povedať, že model mechanizmu RM funguje a je namodelovaný správne. Pri overovaní funkčnosti bolo použitých viac verifikačných dotazov ako bolo uvedených v tejto podkapitole. Pri ich vytváraní sme sa inšpirovali príkladmi z podkapitoly 3.1.5. Z dôvodu rozsahu tejto práce nebudeme podrobne opisovať testovanie ostatných mechanizmov. Avšak testovanie ostatných mechanizmov priebehalo podobne ako v prípade, ktorý sme si opísali v tejto podkapitole.

7 Záver

Cieľom tejto práce bolo vytvorenie simulačných modelov vybraných mechanizmov určených na plánovanie úloh v simulačnom nástroji UPPAAL. Na úvod sme sa zamerali na popis všeobecných poznatkov zameraných na vysvetlenie základných pojmov týkajúcich sa simulácií a modelovania. Následne sme sa venovali predstaveniu samotného nástroja UPPAAL a potom aj jeho rozšíreniu UPPAAL SMC. Postupne sme si vysvetlili jednotlivé konštrukcie tohoto nástroja, od jednoduchších po komplikovanejšie. Najskôr sme sa zamerali na syntaktickú stránku spomenutých konštrukcií a následne aj na vysvetlenie samotného významu na príkladoch modelov, ktoré boli v texte postupne uvedené. Na lepšiu vizualizáciu boli do textu vložené obrázky priamo z nástroja UPPAAL, aby sa čitateľ vedel lepšie orientovať, a aby boli poznatky získané z tejto práce ľahšie využiteľné v praxi.

Na overovanie funkčnosti modelov boli po vytvorení modelov využívané verifikačné dotazy, ktorých syntax a sémantiku sme si vysvetlili v jednej samotnej podkapitole. Pochopenie a vytváranie zložitejších dotazov môže vyžadovať ešte hlbšie rozšírenie znalostí. Nakoľko modely boli vytvárané na princípe fungovania jednotlivých mechanizmov, bola ich funkčnosť vysvetlená postupne po jednom. Každý mechanizmus má iné vlastnosti a každá úloha, ktorá ho využívala bola spustená s parametrami, ktoré sú špecifické pri plánovaní daným mechanizmom. Samotný popis implementácie naväzuje na vysvetlenie základných princípov týchto mechanizmov a ich návrhu v samostatnej kapitole. V texte sú zobrazené aj modely úloh jednotlivých mechanizmov a tak isto aj modely plánovačov, ktoré vykonávanie týchto úloh plánujú.

V poslednej fáze overovania vlastností vytvorených modelov, bol hlavný zámer daný na poukázanie výsledkov na grafoch, ktoré boli vygenerované nástrojom UPPAAL. Pri testovaní sme sa sústredili na využívanie periodických úloh, nakoľko sporadické a aperiodické úlohy by bolo náročnejšie porovnať a vyhodnotiť. Priebeh plánovania týchto úloh bol porovnávaný s výsledkami získanými pomocou nástrojov TimesTool a Cheddar. Toto vyhodnotenie je pre každý mechanizmus zhrnuté v samostatnej podkapitole. Avšak môžeme povedať, že modely vytvorené v nástroji UPPAAL SMC môžeme po porovnaní vyhodnotiť ako zodpovedajúce výsledkom nástrojov UPPAAL a TimesTool. Na týchto výsledkoch sme poukázali na to, kedy je jednotlivé mechanizmy vhodné používať, kedy nie a aké môžu nastať problémy pri plánovaní.

Navrhnuté modely však riešia iba niektoré zo zakladných problémov plánovania úloh v rámci systému. Pokračovaním tejto práce by mohlo byť uvedenie do problematiky pridania prerušovacieho systému do modelov, a následné pozorovanie jeho vplyvu na celkový priebeh úloh. Ďalšou možnosťou by mohlo byť zakomponovanie výpočtovej platformy s už skonštruovanými modelmi či využitie komunikačních mechanizmov na modelovanie nežiadúcich javov, ktoré sa pri plánovaní môžu vyskytnúť (hladovanie, uviaznutie, ...). Pri návrhu a následnej implementácií týchto rozšírení je treba počítať s vyššou náročnosťou na riešenia týchto problémov a predovšetkým s potrebou rozšírenia znalostí vo viacerých smeroch.

Literatúra

(1) **Strnadel, Josef**. *Studijní opora k předmětu ROS. Brno*: FIT VUT v Brně, 2006.

(2) **Peringer, Petr**. *Modelování a simulace. Brno* : FIT/ESF, 2006.

(3) **PENG, Zhaoguang, Yu LU, Alice MILLER, Chris JOHNSON a Tingdi ZHAO**. A Probabilistic Model Checking Approach to Analysing Reliability, Availability, and Maintainability of a Single Satellite System. *2013 European Modelling Symposium* [online]. IEEE, 2013, , 611-616 [cit. 2016-04-16]. DOI: 10.1109/EMS.2013.102. ISBN 978-1-4799-2578-0.
Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6779914

(4) **DUBSLAFF, Clemens, Sascha KLÜPPELHOLZ a Christel BAIER**. Probabilistic model checking for energy analysis in software product lines. *Proceedings of the 13th international conference on Modularity - MODULARITY '14* [online]. New York, New York, USA: ACM Press, 2014, , 169-180 [cit. 2016-05-04]. DOI: 10.1145/2577080.2577095. ISBN 9781450327725. Dostupné z: http://dl.acm.org/citation.cfm?doid=2577080.2577095

(5) **CALINESCU, Radu, Carlo GHEZZI, Kenneth JOHNSON, Mauro PEZZE, Yasmin RAFIQ a Giordano TAMBURRELLI**. Formal Verification With Confidence Intervals to Establish Quality of Service Properties of Software Systems. *IEEE Transactions on Reliability* [online]. 2016, 65(1), 107-125 [cit. 2016-03-21]. DOI: 10.1109/TR.2015.2452931. ISSN 0018-9529.
Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7177126

(6) **MCMILLAN, Kenneth Lauchlin**. *Symbolic Model chcecking: An approach to the state explosion problem* [online]. Pittsburgh, PA, USA, 1992 [cit. 2016-05-9]. Dostupné z: http://www.kenmcmil.com/pubs/thesis.pdf. Ph.D dissertation. Carnegie-Mellon University.

(7) **CHENG, Albert M. K**. *Real-time systems: scheduling, analysis, and verification*. Hoboken, NJ: Wiley-Interscience, 2002. ISBN 0471184063.

(8) **BERNARDO, Marco. a Flavio. CORRADINI**. *Formal methods for the design of real-time systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004 : revised lectures*. New York: Springer, c2004. Lecture notes in computer science, 3185. ISBN 978-3-540-23068-7.

(9) **DAVID, Alexandre, Kim G. LARSEN, Axel LEGAY, Marius MIKUČIONIS a Danny Bøgsted POULSEN**. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer* [online]. 2015, **17**(4), 397-415 [cit. 2016-02-12]. DOI: 10.1007/s10009-014-0361-y. ISSN 1433-2779. Dostupné z: http://link.springer.com/10.1007/s10009-014-0361-y

(10) **ALUR, Rajeev a David L. DILL**. A theory of timed automata. *Theoretical Computer Science* [online]. 1994, **126**(2), 183-235 [cit. 2016-03-22]. DOI: 10.1016/0304-3975(94)90010-8. ISSN 03043975. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/0304397594900108

(11) **Strnadel, Josef**. Návrh časově kritických systémů I: specifikace a verifikace. *Automa*. 2010, roč. 2010, č. 10, s. 42-44. ISSN 1210-9592.

(12) **Strnadel, Josef**. Návrh časově kritických systémů II: úlohy reálného času. *Automa*. 2010, roč. 2010, č. 12, s. 18-19. ISSN 1210-9592.

(13) **COTTET, Francis., Joëlle. DELACROIX a Zoubir. MAMMERI**. *Scheduling in real-time systems* [online]. Hoboken, NJ, USA: J. Wiley, c2002 [cit. 2016-02-04]. ISBN 04-708-4766-2.

(14) **Strnadel, Josef**. Návrh časově kritických systémů III: priorita úloh. *Automa*. 2011, roč. 2011, č. 2, s. 50-52. ISSN 1210-9592.

Zoznam príloh

Príloha 1

Obsah CD

- /src/ -zdrojové súbory modelov
  - /UPPAAL_modely/ - .xml súbory pre UPPAAL
  - /TimesTool_modely/ - .xml súbory pre TimesTool
  - /Cheddar_modely/ - .xml súbory pre Cheddar
- /tools/ - súbory na spustenie jednotlivých nástrojov
  - /uppaal-4.1.19/ - súbory na spustenie nástroja UPPAAL
  - /Timestool/ - súbory na spustenie nástroja TimesTool
  - /Cheddar21/ - súbory na spustenie nástroja Cheddar
- /doc/ - tento dokument vo formáte docx a pdf
- manual.pdf - používateľská príručka k tejto práci
- README